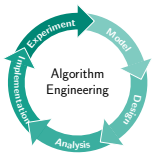


Master Thesis

A Multilevel Design for Scalable Pareto-Optimal Public Transit Queries

Patrick Steil

5 May – 5 November 2025

Examiner: Prof. Dr. Peter Sanders
Prof. Dr. Thomas Bläsius
Supervisors: Dr. Jonas Sauer
M.Sc. Sascha Witt

Institute of Theoretical Informatics, Algorithm Engineering
Department of Informatics
Karlsruhe Institute of Technology

Abstract

English

This thesis investigates routing algorithms for public transit networks. Especially in the context of countrywide or continental-sized transit networks, current state-of-the-art algorithms reach their limits: Fast query times can usually only be achieved through time-consuming preprocessing and high memory overhead.

Until now, no algorithm has been able to efficiently compute Pareto-optimal journeys with respect to minimal arrival time and minimal number of transfers with both low memory consumption and minimal preprocessing effort on large networks.

We present T-REX (Transfer-Ranked EXploration), a new algorithm that narrows this gap by combining the state-of-the-art Trip-Based Public Transit Routing Algorithm with multilevel partitioning techniques.

The underlying idea is to partition stops into multilevel cells and identify relevant transfers for long-distance travel during a short precomputation phase. T-REX then uses this precomputed information to restrict the search space during a query, effectively pruning unnecessary local transfers.

Unlike previous partition-based approaches, T-REX has two key advantages. Firstly, the query phase operates directly on the existing graph structure, without the need to build any additional data structures. Secondly, as T-REX focuses on transfers rather than lines or trips, it can directly determine which events are worth exploring next, avoiding the overhead of computing the next reachable events.

In this work, T-REX is described in detail, compared with existing methods, and evaluated in extensive experiments on real public transit data.

The algorithm is able to handle even large networks such as the whole of Europe with preprocessing times of a few minutes. At the same time, it allows for query times with an average runtime of around $18\,\mathrm{ms}$, compared to $222\,\mathrm{ms}$ achieved by previous state-of-the-art algorithms, and requires only $515\,\mathrm{MB}$ additional memory compared to TB.

These properties make T-REX particularly suitable for use in interactive real-time applications, e.g., in mobile navigation services.

Deutsch

Diese Arbeit beschäftigt sich mit Routingalgorithmen im öffentlichen Personennahverkehr (ÖPNV). Gerade bei landesweiten oder kontinentalen Netzwerken stoßen aktuelle State-of-the-art-Algorithmen an ihre Grenzen: Schnelle Anfragezeiten lassen sich dort meist nur durch immense Vorverarbeitung und hohen Speicherverbrauch erzielen.

Ein Algorithmus, der bei geringem Speicherbedarf und kurzer Vorverarbeitungszeit schnell Pareto-optimale Lösungen hinsichtlich Ankunftszeit und Anzahl der Umstiege berechnet, war bisher nicht bekannt.

Wir präsentieren T-REX (Transfer-Ranked EXploration), einen neuen Algorithmus, der diese Lücke adressiert, indem er den state-of-the-art Trip-Based Public Transit Routing Algorithm [1] mit mehrstufigen Partitionierungstechniken kombiniert.

Die zugrunde liegende Idee besteht darin, die Haltestellen in mehrere Ebenen von Zellen zu partitionieren und relevante Umstiege für Fernreisen während einer kurzen Vorberechnungsphase zu markieren. T-REX nutzt diese vorberechneten Informationen dann, um den Suchraum während einer Anfrage einzuschränken und so unwichtige lokale Bereiche nicht absuchen zu müssen. Im Gegensatz zu bisherigen partitionierungsbasierten Ansätzen bietet T-REX zwei zentrale Vorteile. Erstens arbeitet die Anfrage direkt auf dem vorhandenen Graphen, ohne dass zusätzliche Datenstrukturen aufgebaut werden müssen. Zweitens konzentriert sich T-REX auf Umstiege zwischen einzelnen Ereignissen anstatt auf ganze Linien oder Fahrten. Dadurch kann der Algorithmus effizient von einem Ereignis zum nächsten fortschreiten, ohne dafür zusätzlichen Aufwand investieren zu müssen.

T-REX wird detailliert beschrieben, mit bestehenden Algorithmen verglichen und im Rahmen umfangreicher Experimente auf realen ÖPNV-Daten evaluiert.

Der Algorithmus ist in der Lage, selbst große Netze wie ganz Europa innerhalb weniger Minuten vorzuberechnen. Gleichzeitig ermöglicht er Anfragen mit einer Laufzeit von durchschnittlich nur etwa $18\,\mathrm{ms}$ und benötigt dabei lediglich $515\,\mathrm{MB}$ mehr Speicherplatz als TB.

Mit diesen Eigenschaften ist T-REX besonders gut für den Einsatz in interaktiven Echtzeitanwendungen geeignet, etwa in mobilen Navigationsdiensten.

Acknowledgments

I am very grateful to Professor Dr. Peter Sanders, who supervised my Master's thesis and shaped my understanding of algorithm design throughout my studies.

I feel fortunate to have learned from so many people, both through direct discussions and through their contributions to the field. I would like to express my deep gratitude to everyone who helped lay the foundations of modern route planning algorithms and shared their insights along the way. These conversations and the underlying research have taught me a great deal and inspired new ideas. These include Dr. Ben Strasser, Dr. Lars Gottesbüren, Sascha Witt, Moritz Laupichler, Dr. Lukas Barth, and especially Dr. Jonas Sauer. Jonas has consistently emphasized that naming algorithms should be a central part of research and not just random concatenations of characters.

I would also like to thank Professor Dr. Christian Schulz for providing the original implementations of KaHIP and Buffoon, Dr. Ben Strasser for the original implementations of CSA and ACSA, and Dr. Felix Gündling and Dr. Patrick Brosi for supplying the datasets that were essential for our experiments.

Finally, and most importantly, I am deeply grateful to my friends and family, who have always given me motivation, joy, and the opportunity to pursue what I love.

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Ort, den November 3, 2025

Contents

Abstract iii

1 Introduction 1
1.1 Motivation . . . 1
1.2 Contribution . . . 2
1.3 Structure of Thesis . . . 3

2 Related Work 5

3 Fundamentals 9
3.1 Public Transit Network . . . 9
3.2 Graph . . . 12
3.3 Modeling as Graph . . . 13
3.3.1 Time-Dependent Graph . . . 14
3.3.2 Time-Expanded Graph . . . 15
3.4 Transfer Patterns . . . 16
3.4.1 Scalable Transfer Patterns . . . 17
3.5 Connection Scan Algorithm . . . 18
3.5.1 Partitioning the Connections . . . 20
3.6 Trip-Based Public Transit Routing . . . 21
3.6.1 Transfer Precomputation . . . 21
3.6.2 Fixed Departure Time Query Algorithm . . . 23
3.6.3 Journey Unpacking . . . 25
3.6.4 Profile Query Algorithm . . . 25
3.6.5 Event-to-Event Query . . . 26

4 Transfer-Ranked EXploration 27
4.1 Introduction . . . 27
4.2 Graph Representation . . . 28
4.3 Partitioning . . . 29
4.4 Query . . . 30
4.5 Customization . . . 31
4.5.1 Partial Proof of Correctness . . . 32
4.5.2 Parallelization . . . 38

4.6 Advanced Techniques and Extensions . . . 39
4.6.1 Bit Manipulation . . . 39
4.6.2 Transfer Shortcuts . . . 40
4.6.3 Delays and Cancellations . . . 41

5 Experiments **43**
5.1 Datasets . . . 43
5.2 Preparation . . . 48
5.3 Graph Partitioner . . . 49
5.4 Customization . . . 50
5.4.1 Efficiency . . . 51
5.4.2 Transfer Distribution . . . 52
5.5 Query . . . 57
5.5.1 Methodology . . . 57
5.5.2 Fixed Departure Time Queries . . . 58
5.5.3 Fixed Departure Time Geo-Rank Queries . . . 60
5.5.4 Profile Queries . . . 61
5.6 Comparison With Other Algorithms . . . 63
5.6.1 ACSA . . . 63
5.6.2 Graph-Based Algorithms . . . 65
5.6.3 FLASH-TB and TB-CST . . . 68
5.6.4 (Scalable) Transfer Patterns . . . 69
5.6.5 Comparison to CRP . . . 70
5.7 Memory–Query–Preprocessing Trade-Offs . . . 74
5.7.1 Memory Consumption . . . 74
5.7.2 Precomputation Time . . . 74

6 Conclusion **77**
6.1 Future Work . . . 77

A More Experiments **79**
A.1 T-REX Metrics . . . 79
A.2 Hub Labeling Experiments . . . 83

Bibliography **85**

1 Introduction

Before we discuss the details of the current state-of-the-art algorithms, it is important to first understand why efficient route planning in public transport is both challenging and relevant. In this chapter, we therefore provide the necessary background and context, outline the motivation behind this work and summarize its main contributions. This lays the groundwork for the subsequent chapters, in which we present the main work of this thesis: T-REX.

1.1 Motivation

We encounter journey planning in public transport every day, whether on the morning commute to work or on vacation across countries. Every day, millions of travel requests are sent to systems that compute optimal journeys. These include services such as Google Maps[1] or official websites of local or national transit agencies.

Unlike routing for individual transport, such as driving, where users typically optimize a single criterion like travel time or distance, travelers using public transport often wish to optimize several criteria all together [2]. These include arrival time, of course, but also convenience, such as the number of transfers between vehicles. Therefore, modern journey planning algorithms typically compute Pareto-optimal journeys in terms of arrival time and number of transfers, minimizing both criteria [2, 3, 4]. In order to be able to answer such queries quickly and efficiently, many state-of-the-art algorithms precompute important information that speeds up the search. This type of technique is called *speedup technique* [5, 6, 7].

When assessing the usability and performance of routing algorithms, operators consider several metrics, such as query performance, memory consumption, precomputation time and the incorporation of real-time information. For road networks, algorithms such as Customizable Route Planning (CRP) [8] or Customizable Contraction Hierarchies (CCH) [9] have proven to perform well across all of these metrics. By contrast, despite extensive research, there is still no single journey planning algorithm for public transit that achieves comparable performance across all these aspects. Consequently, when choosing an algorithm, one has to trade off between query performance, preprocessing time, and real-time capability. An approach that simultaneously offers sub-millisecond query times, manageable preprocessing, and low memory overhead has not yet been found.

[1] `https://maps.google.com/`

1.2 Contribution

In this thesis, we present T-REX (Transfer-Ranked EXploration), a new public transit journey planning algorithm that enables interactive query times even on continent-sized networks while maintaining moderate preprocessing times and memory consumption.

T-REX combines ideas from TB (Trip-Based Public Transit Routing) and multilevel graph partitioning techniques known from road network routing, such as Customizable Route Planning (CRP). Given a timetable, T-REX partitions the stops into cells across multiple levels and then computes information about which transfers between trips are required for long-distance travel. Then, during a query, this information allows the search to skip unimportant cells, significantly reducing the search space without requiring heavy preprocessing.

We benchmarked T-REX against a wide range of state-of-the-art public transit routing algorithms, using either the original implementations or reimplementations based on the stated publications. All algorithms were evaluated on identical datasets and hardware setups, covering networks of various scales; from metropolitan areas (e.g., Paris) to entire countries (e.g., Germany) and the full European dataset.

On the European instance, a dataset with over 1.3 million stops and over 100 million events, T-REX can answer a query in about $18\,\mathrm{ms}$, requiring less than 10 minutes of precomputation time. TB [1], RAPTOR [3] and CSA [10] need an average of around 200, 715 and $390\,\mathrm{ms}$, respectively, for a query.

Unlike previous algorithms that either require extensive preprocessing (often taking hours or days [5, 6]) to achieve sub-milliseconds queries or perform almost no preprocessing but suffer from slower queries (such as RAPTOR [3] or CSA [10]), T-REX occupies a good Pareto spot: preprocessing takes only a few minutes even for large continental networks, while query times remain within a few milliseconds.

Two partition-based algorithms, that also achieve reasonable precomputation times, are HypRAPTOR [11] and ACSA [10]. Both approaches exhibit certain limitations. HypRAPTOR partitions the network and stores which lines are important for optimal travel across cells. Since it stores information about lines rather than about individual events, the algorithm still needs to process many irrelevant events during queries. ACSA, on the other hand, works similarly to T-REX with a multilevel partition, but has to build additional data structures for a query, which costs valuable time. Further details on both algorithms are provided in Chapter 4.

While T-REX achieves an attractive trade-off between preprocessing and query performance, the handling of real-time updates remains an open challenge. Because some transfers must be regenerate as soon as schedules change, parts of the preprocessing would need to be repeated. However, this “update” step would likely be significantly faster than a full recomputation, although we did not perform dedicated experiments to verify this.

Another current limitation of T-REX is the efficiency of preprocessing for very long or irregular timetables, as preprocessing currently scales linearly with the number of extracted days. This needs further investigation.

1.3 Structure of Thesis

This Master's thesis is structured as follows:

- Chapter 2 provides an overview of existing routing algorithms in public transit.
- Chapter 3 introduces key definitions and presents state-of-the-art algorithms in detail, laying the groundwork for the discussion of T-REX in the following chapter.
- Chapter 4 describes the T-REX algorithm in detail.
- Chapter 5 presents experimental results, comparing T-REX to state-of-the-art algorithms in terms of precomputation time, memory overhead and query performance.
- Finally, Chapter 6 summarizes all the results relating to T-REX and provides an outline of possible future research directions.

2 Related Work

In the following, we consider journey planning algorithms that either compute the earliest arrival time or Pareto-optimize arrival time and number of vehicles used. Acceleration techniques that enable efficient queries on public transport networks have long been a problem [2, 12]. Typical approaches to find optimal journeys are based on modeling the timetable as a graph and traversing this graph at query time.

Time-Dependent Graph. In the *time-dependent* model, vertices represent the stops, and edges between stops hold time functions which, e.g., map departure time to arrival time or departure time to journey duration. A shortest path algorithm maintains the earliest possible arrival time for each stop and must evaluate the corresponding time function when relaxing an edge and thus compute the new arrival time. This evaluation step costs more time than (as usual) adding two scalar values. One such algorithm that can handle these time functions is TDD [13] (Time-Dependent Dijkstra). In order to guide the search through the network in a more targeted manner, there exist A^* extensions [13].

Time-Expanded Graph. The second approach to modeling the timetable as a graph is the *time-expanded* model. Here, the time dimension is "expanded", i.e., the time information is represented by the structure of the graph. Events that take place at a certain point in time at a certain location are modeled as vertices, and an edge from u to v represents that starting from u, one can reach v. To answer earliest arrival queries, the query algorithm starts at the first reachable vertex at the source stop and scans through the graph until the first reachable vertex at the target stop is found. One such algorithm is TED [12] (Time-Expanded Dijkstra), which in contrast to TDD does not need to evaluate complex time-functions when relaxing an edge, as all edge weights are scalar. This approach has some disadvantages, due to the graph size. It is much bigger than the time-dependent one, and real-time data must be integrated into the graph in a more complicated manner, especially in comparison to the time-dependent model. However, since it uses a standard Dijkstra algorithm [14], it is conceptually simpler and easier to implement. For more details about the two approaches, their performance and several speedup techniques, we refer to [15]. The fastest query times are achieved by PTL [16] (Public Transit Labeling), an algorithm that precomputes all shortest paths between all pair of vertices on the time-expanded graph and stores them in a very compact way, analogous to Hub Labeling. Hub Labeling [17] is a state-of-the-art routing algorithm for shortest path queries on graphs, but it also has disadvantages (which PTL also inherits). Firstly, the memory consumption is relatively high, despite the compact

representation of the data, and secondly, the algorithm is a distance oracle, i.e., it only returns the shortest distance between two vertices. It should also be mentioned that the memory consumption of PTL for Pareto-optimal queries is significantly higher compared to the memory consumption for "only" earliest arrival queries.

Timetable based. Algorithms that bypass the graph modeling step act directly on the timetable. These include e.g., RAPTOR [3] and CSA [10], which take advantage of modern hardware through better cache efficiency to outperform graph-based algorithms. RAPTOR [3] (Round-bAsed Public Transit Optimized Router) is based on an idea similar to a breadth-first search, whereby a round is equivalent to sitting in a vehicle. In each round, the algorithm tries to minimize the arrival time at the destination stop, and over all rounds all Pareto-optimal journeys are found. TB [1] (Trip-Based Public Transit Routing) is similar to RAPTOR in the sense that the network is explored in rounds at query time and thus Pareto-optimal solutions are found. However, instead of jumping from line to line, TB pre-calculates important transfers between trips and then hops from trip to trip. Precomputing these transfers is a matter of minutes, even on large networks, and is therefore feasible in practice. Query times are faster than RAPTOR, at the expense of a slight memory overhead for the transfers. CSA [10] (Connection Scan Algorithm) is a relatively simple algorithm which keeps all elementary connections sorted according to their departure time in an array. The query itself iterates over these connections, similar to a dynamic program, and minimizes the arrival times for all reachable stops. This rather simple idea (*iterating over a sorted array*) arose from the realization that the time-expanded graph is directed and acyclic, and thus shortest paths can be found in linear time. This only requires a topological order, which is given in the time-expanded graph by sorting all elementary connections by their timestamps. This means that a graph data structure is no longer required, only an array is sufficient for the linear sweep. It is important to mention that CSA, in its basic version, only minimizes the arrival time and has to pay for Pareto-optimal results with a worse runtime.

In order to achieve even faster query times in the submillisecond range, algorithms previously had to carry out a quadratic precomputation effort, which is no longer acceptable, especially for country-sized networks.

(Scalable) Transfer Patterns. Another well-known approach is Transfer Patterns [6]. The algorithm stores all the necessary transfer stops and the lines used to answer a query for each pair of stations. Queries can be answered very quickly because the search space consists only of the precomputed stations and lines, and is extremely small. As this requires an enormous amount of memory, Scalable Transfer Patterns [18] divides the underlying network into many small local clusters and computes local and global transfer patterns. Local transfer patterns "connect" stations of a cluster to other inner stations and borderstops, and global transfer patterns "connect" borderstops across clusters. A query must first assemble the search space from both local and global transfer patterns, which introduces additional

overhead and results in query times only slightly better than those of TB. Unfortunately, Scalable Transfer Patterns only solves the memory problem, not the problem of expensive precomputation. Basically, the precomputation takes the same amount of time, as both algorithms have to answer every possible query during the precomputation. In order to reduce the precomputation time, both algorithms drop provable correctness. However, it is shown that such "optimizations" are useful in practice.

TB-based approaches. Based on the underlying idea of Transfer Patterns, TB-CST [7] (TB using Condensed Search Trees) achieves similar query times as Transfer Patterns, but the precomputation is significantly faster and the memory consumption is lower (but still too high for any practical application). The precomputation still solves all possible queries, but the algorithm is based on TB, not Dijkstra's algorithm; and hence faster. TB-CST saves a tree for each stop, which has the stop as the root, lines as inner vertices and the reachable stations as leaves. A query only scans the precomputed search tree of the start stop, and is very fast due to the very small search space. To reduce memory consumption, these trees are split into prefix and suffix trees and joined together during the query. Unfortunately, this results in slower queries, and the splicing of the suffix- and prefix trees dominate the runtime, especially on metropolitan instances such as London. Another TB-based state-of-the-art algorithm is FLASH-TB [5] (FLAgged SHortcuts-TB). Instead of running the query on additional data structures such as trees or lookup tables, the FLASH-TB query scans through the same graph as TB, but in a more goal-oriented manner. The set of stops is partitioned into k cells, and at each transfer k bits mark for which cell the transfer is "important". More precisely: The i-th bit is set to true if this transfer is on an optimal path to a stop of the i-th cell. The query therefore only relaxes the transfers that are important for the target cell. The search space is so small that the query times of Transfer Patterns and TB-CST are beaten. However, the precomputation is also expensive in the sense that all possible queries have to be answered in advance. Memory consumption is linear in size and thus very low, which was one of the major problems with Transfer Patterns and TB-CST.

Partition based. All the "heavy" precomputation-based algorithms do not allow for fast real-time updates, such as delays or failures. Experimentally, Transfer Patterns have been shown to be relatively robust to delays [19]. However, as only delay scenarios were evaluated, it is unclear how Transfer Patterns can handle events such as blocked tracks or closed stations due to e.g., signal box malfunctions.

An algorithm that allows a relatively small "update" phase is ACSA [10] (Accelerated Connection Scan Algorithm). CSA scans a large number of unnecessary connections, e.g., when a local query is made in Munich, non-reachable connections in Berlin are also scanned. In order to give the algorithm an understanding of "local" and "global" connections, ACSA partitions the underlying network into multilevel cells and precomputes the connections per level that are important for traversing cells. A query only considers connections that either lie within the local cell of the source or target stop, or bypass

"unimportant" cells. The query performance is slower than the algorithms with quadratic precomputation, but the approach seems promising for future work as it achieves query times of some milliseconds on country-sized networks while having a small memory overhead. In addition, the importance of connections can be precomputed in a few minutes, which makes ACSA much more attractive for real-time applications. Unfortunately, ACSA is only evaluated for earliest arrival queries, which is a much simpler problem. Other notable examples of acceleration techniques based on partitioning are HypRAPTOR [11] and its TB variant HypTB [20]. Unlike FLASH-TB or ACSA, HypRAPTOR models the network as a hypergraph, where stops are hyperedges and lines are pins. This hypergraph is partitioned and then *fill-in* lines are calculated, which are required for cross-cell queries. Unfortunately, the hypergraph-based approaches have no notable success: HypRAPTOR reported a speedup of two, while HypTB is even slower than the normal TB on some instances.

In addition, acceleration techniques for shortest paths in individual transport, e.g., on road networks, which are based on partitions, are very successful. Road graphs have very small balanced separators and can therefore be partitioned very efficiently, see e.g., [21]. Techniques such as CRP [8] (Customizable Route Planning) or CCH [9] (Customizable Contraction Hierarchies) achieve speedups of several orders of magnitude compared to the Dijkstra baseline, and allow updating edge metrics in a few seconds on continent-sized networks. Applying these techniques to public transportation networks seem unsuccessful because, among other things, the underlying graph properties (of a road graph) are lost and therefore the algorithms reach their limits.

3 Fundamentals

This chapter introduces basic definitions that form the basis of the various routing algorithms, and explains the main approaches and algorithms in detail to provide an overview.

In this thesis, unless stated otherwise, all intervals (closed, open, or half-open) only contain natural numbers. Hence, an interval I denotes the set $I \cap \mathbb{N}^+$.

3.1 Public Transit Network

We follow notation from [5]. A *public transit network* is a 6-tuple $(\mathcal{S}, \mathcal{F}, \mathcal{E}, \mathcal{T}, \mathcal{L}, \Phi)$ consisting of a set of stops $\mathcal{S}$, footpaths $\mathcal{F} \subseteq \mathcal{S} \times \mathcal{S}$, events $\mathcal{E}$, trips $\mathcal{T}$, lines $\mathcal{L}$ and a service period $\Phi \subseteq \mathbb{N}$. The service period Φ defines the time period during which the trips operate. In this thesis, we represent times within the service period in seconds, i.e., each point in time is given as the number of seconds elapsed since midnight of the first day. Each element in the sets $\mathcal{S}, \mathcal{T}, \mathcal{E}$, and $\mathcal{L}$ is assigned a unique *identifier* ID, which is an integer in a consecutive range starting from 0. That is, we define the following mappings:

$$\begin{aligned}
\mathrm{ID}_{\mathcal{S}} &: \mathcal{S} \to \{0, \ldots, |\mathcal{S}| - 1\} \\
\mathrm{ID}_{\mathcal{T}} &: \mathcal{T} \to \{0, \ldots, |\mathcal{T}| - 1\} \\
\mathrm{ID}_{\mathcal{E}} &: \mathcal{E} \to \{0, \ldots, |\mathcal{E}| - 1\} \\
\mathrm{ID}_{\mathcal{L}} &: \mathcal{L} \to \{0, \ldots, |\mathcal{L}| - 1\}
\end{aligned}$$

For convenience, we sometimes refer to a stop, trip, or line either by the element itself (e.g., $s \in \mathcal{S}$) or by its corresponding ID (e.g., $\mathrm{ID}_{\mathcal{S}}(s),\ s \in \mathcal{S}$), depending on the context.

A *stop* $p \in \mathcal{S}$ is a geographical point at which passengers can embark and disembark vehicles, and a journey of a vehicle is defined as *trip* $T \in \mathcal{T}$. A trips visit at a stop is called a *stop event* $\varepsilon \in \mathcal{E}$, and is represented as a 4-tuple with the associated trip T, the stop $p(\varepsilon) \in \mathcal{S}$ and arrival $\tau_{\mathrm{arr}}(\varepsilon) \in \Phi$ and departure time $\tau_{\mathrm{dep}}(\varepsilon) \in \Phi$. A trip is defined as a sequence of stop events, i.e., $T = \langle \varepsilon_1, \varepsilon_2, \ldots \rangle$, where the number of stop events of the trip is written as $|T|$. In a straightforward way, $T[i]$ denotes the i-the stop event, and $T[i, j]$ for $1 \leq i < j \leq |T|$ is the trip segment between (and including) the i-th and j-th stop event. For simplicity, for a trip $T \in \mathcal{T}$, we set $\tau_{\mathrm{arr}}(T[1]) \coloneqq \tau_{\mathrm{dep}}(T[1])$ and $\tau_{\mathrm{dep}}(T[|T|]) \coloneqq \tau_{\mathrm{arr}}(T[|T|])$. Additionally, we assume that no consecutive events of any trip occur at the same time:

$$\forall T \in \mathcal{T} \, \forall i \in [1, |T|) : \tau_{\mathrm{arr}}(T[i]) < \tau_{\mathrm{arr}}(T[i+1]) \wedge \tau_{\mathrm{dep}}(T[i]) < \tau_{\mathrm{dep}}(T[i+1])$$

We require that trips are well defined and prohibit time travel, i.e., for all trips $T \in \mathcal{T}$:

$$\forall\, i \in [1, |T|] : \tau_{\text{arr}}\left(T[i]\right) \leq \tau_{\text{dep}}\left(T[i]\right) \text{ and}$$
$$\forall\, i \in [1, |T|) : \tau_{\text{dep}}\left(T[i]\right) \leq \tau_{\text{arr}}\left(T[i+1]\right)$$

A trip $T \in \mathcal{T}$ is made of $|T|-1$ *connections*, with one connection containing the information of a trip segment of length two $T[i, i+1]$, $i \in [1, |T|)$. Formally, a connection c is a 5-tuple

$$c = \left(\text{ID}_{\mathcal{T}}\left(T\right), \tau_{\text{dep}}\left(T[i]\right), p\left(T[i]\right), \tau_{\text{arr}}\left(T[i+1]\right), p\left(T[i+1]\right)\right).$$

For simplicity, we overload $\tau_{\text{dep}}\left(c\right)$ and $\tau_{\text{arr}}\left(c\right)$ to refer to the departure and arrival time of the underlying trip segment. Also, we denote by $p_{\text{arr}}\left(c\right)$ and by $p_{\text{dep}}\left(c\right)$ the arrival and departure stop, as well as by $T(c)$ the underlying trip. The set of all connections is referred to as $\mathcal{C}$. We indicate the sequence of stops that are visited by a trip with

$$\Pi\left(T\right) = \langle p\left(T[i]\right) \mid \forall\, i \in [1, |T|]\rangle.$$

Trips are grouped into *lines* $\mathcal{L}$, whereby two trips on the same line must follow the same stop sequence and may not overtake each other. This property is called FIFO, and lines are FIFO-orderings of trips. We define the stop sequence of a line $L \in \mathcal{L}$ as the stop sequence of a representative trip of the line, written as $\Pi\left(L\right)$. Again, for simplicity, we assume no two trips of the same line are equal, i.e., $\forall\, L \in \mathcal{L}$:

$$\forall\, T_a \neq T_b \in L\, \exists\, i \in [1, |\Pi\left(L\right)|) : \tau_{\text{arr}}\left(T_a[i]\right) \neq \tau_{\text{arr}}\left(T_b[i]\right) \wedge \tau_{\text{dep}}\left(T_a[i]\right) \neq \tau_{\text{dep}}\left(T_b[i]\right)$$

For more information on the FIFO-ordering of trips into lines, we refer to [5]. A *footpath* $(p, q) \in \mathcal{F}$ connects two stops $p, q \in \mathcal{S}$ and encodes walking in a non-negative time $\Delta\tau_{\text{fp}}(p, q)$ between from p to q. We require that the set of footpaths is transitively closed, meaning

$$\forall p, q, r \in \mathcal{S} :\ (p, q), (q, r) \in \mathcal{F} \implies (p, r) \in \mathcal{F},$$

and fulfills the triangle inequality, i.e.,

$$\forall p, q, r \in \mathcal{S} :\ \Delta\tau_{\text{fp}}(p, r) \leq \Delta\tau_{\text{fp}}(p, q) + \Delta\tau_{\text{fp}}(q, r).$$

For simplicity, we denote by $\mathcal{F}^{\uparrow}(s)$ the set of stops reachable by outgoing footpaths of stop $s \in \mathcal{S}$, and $\mathcal{F}^{\downarrow}(s)$ the set of stops from which a footpath towards s begins.

A p_s-p_t-*journey* $J = \langle f_0, T_1[i_1, j_1], \mathfrak{t}_1, \ldots, \mathfrak{t}_{k-1}, T_k[i_k, j_k], f_{k+1}\rangle$ describes a voyage from the source stop p_s to the target stop p_t through the network and consists of initial and final footpaths $f_0, f_{k+1} \in \mathcal{F}$ such that $f_0 = (p_s, p\left(T_1[i_1]\right))$ and $f_{k+1} = \left(p\left(T_k[j_k]\right), p_t\right)$, and alternating trip segments and transfers in between, whereby a transfer $\mathfrak{t}_l = (T_l[j_l], T_{l+1}[i_{l+1}])$ connects two consecutive trip segments. A transfer $\mathfrak{t} = (T_a[j], T_b[i])$ is based on a footpath $f = \left(p\left(T_a[j]\right), p\left(T_b[i]\right)\right) \in \mathcal{F}$ such that the transfer time $\Delta\tau_{\text{fp}}(p\left(T_a[j]\right), p\left(T_b[i]\right))$ allows to reach $T_b[i]$ from $T_a[j]$ in time, i.e.,

$$\tau_{\text{arr}}\left(T_a[j]\right) + \Delta\tau_{\text{fp}}(p\left(T_a[j]\right), p\left(T_b[i]\right)) \leq \tau_{\text{dep}}\left(T_b[i]\right).$$

Note that we omit empty initial or final footpaths from the journey notation. We use $|J|$ to describe the number of trip segments in J, and thus $|J| - 1$ corresponds to the number of transfers (if $|J| > 0$).

In some datasets, stations with multiple tracks are represented as a single stop. To account for the additional time required to transfer between tracks (within the station), a *minimum change time* can be introduced for a stop $p \in \mathcal{S}$.

In this work, we do not model this time explicitly but instead incorporate it implicitly, following the approach of [22].

For each stop $p \in \mathcal{S}$, we define a *buffer time* $\tau_{\text{buf}}(p) \geq 0$, which specifies the waiting time at stop p to board a trip. We modify each event $\varepsilon \in \mathcal{E}$ by subtracting the buffer time from the departure time, i.e., $\tau_{\text{dep}}(\varepsilon) \leftarrow (\tau_{\text{dep}}(\varepsilon) - \tau_{\text{buf}}(p(\varepsilon)))$. This may result in departure times being earlier than the corresponding arrival times for certain events. However, since the departure time is only relevant when boarding a trip, and not while already traveling, the correctness of the algorithms remains unaffected.

For two events $\varepsilon_{\text{s}}, \varepsilon_{\text{t}} \in \mathcal{E}$, we define a ε_{s}-ε_{t}-journey $J_{\mathcal{E}} = \langle T_1[i_1, j_1], \mathfrak{t}_1, \ldots, \mathfrak{t}_{k-1}, T_k[i_k, j_k] \rangle$, which analogously to the stop-to-stop journey describes a path through the network, but starting at $\varepsilon_{\text{s}} = T_1[i_1]$ and ending at $\varepsilon_{\text{t}} = T_k[j_k]$. Given a journey J and indices $1 \leq m < n \leq k$, we define the event-to-event subjourney $J[T_m[i_m], T_n[j_n]]$ as $T_m[i_m]$-$T_n[j_n]$ journey using the same transfers and trips as J.

The following definitions are based on [5] and [22].

Given a query q with a source stop $p_s \in \mathcal{S}$, a target stop $p_t \in \mathcal{S}$ and a departure time $\tau_{\text{dep}} \in \Phi$. A p_s-p_t-journey J is *feasible* for q if J is a p_s–p_t journey departing no earlier than τ_{dep}. Let $\mathcal{J}$ be the set of feasible journeys.

We define a *criterion* $c_i : \mathcal{J} \mapsto C_i$ as a function that maps a given journey $J \in \mathcal{J}$ to a *cost space* C_i, with $i \in \mathbb{N}$. If multiple criteria $\mathcal{C} = (c_1, c_2, \ldots)$ are given, we define the *cost vector* $c(J)$ for a given journey J as

$$c(J) = (c_1(J), c_2(J), \ldots, c_i(J)) .$$

For given criteria $\mathcal{C} = (c_1, c_2, \ldots)$, e.g., minimizing arrival time and minimizing the number of transfers, we say a journey J *weakly dominates* another journey J' if no criteria of J is worse than of J'. Additionally, we say J *strongly dominates* J' if J weakly dominates J' and there is at least one criteria for which J is better than J'.

A feasible journey J is thus *Pareto-optimal* for q if there does not exist another feasible journey J' which strongly dominates J. Because for given criteria $\mathcal{C} = (c_1, c_2, \ldots)$, there could be many Pareto-optimal journeys, we define the *Pareto-front* $\mathfrak{F}$ as

$$\mathfrak{F} = \{c(J) \mid J \in \mathcal{J} \wedge J \text{ is Pareto-optimal}\} \subseteq C_1 \times C_2 \times \cdots \times C_i.$$

Additionally, we define the *representative set* $\mathfrak{J} \subseteq \mathcal{J}$ as a minimal set of feasible journeys, such that $\mathfrak{F} \subseteq \{c(J) \mid J \in \mathfrak{J}\}$.

In this thesis, we consider different types of queries. Let $p_s, p_t \in \mathcal{S}$, $\varepsilon_\mathrm{s}, \varepsilon_\mathrm{t} \in \mathcal{E}$ and $\tau, \tau_\mathrm{start}, \tau_\mathrm{end} \in \Phi$ be given.

Definition 3.1.1 (Fixed Departure Time Query)**.** *Given optimization criteria, a fixed departure time query $q = (p_s, p_t, \tau)$ asks for a representative set $\mathfrak{J} = \{J_1, J_2, \dots\}$ with respect to the given criteria.*

Definition 3.1.2 (Profile Query)**.** *Given optimization criteria, a profile query $q = (p_s, p_t, [\tau_\mathrm{start}, \tau_\mathrm{end}])$, with $\tau_\mathrm{start} \leq \tau_\mathrm{end}$, asks for a representative set $\mathfrak{J} = \{J_1, J_2, \dots\}$ with respect to the given criteria and maximizing departure time.*

Definition 3.1.3 (Event-To-Event Query)**.** *An event-to-event query $q = (\varepsilon_\mathrm{s}, \varepsilon_\mathrm{t})$ asks for a ε_s-ε_t-journey $J_\mathcal{E}$ which minimizes the number of transfers.*

Throughout this thesis, unless stated otherwise, we always consider the given criteria to be minimizing arrival time and the number of transfers.

3.2 Graph

Let O be a set and $k \in \mathbb{N}^+$. We define $\mathcal{H}(O) = \{H_0, H_1, \dots, H_{k-1}\}$ as a *k-partition* of O iff $\bigcup_{i\in[0,k)} H_i = O$ and $\forall\, i \neq j \in [0,k) :\ H_i \cap H_j = \emptyset$. A set $H \in \mathcal{H}(O)$ is called a *cell*. If $k = 2$, we denote by $\mathcal{H}^2(O)$ a *bipartition* of O.

In practice, an additional *imbalance* parameter $\varepsilon \in \mathbb{R}^+$ is introduced for k-partitions, which ensures that all cells $H_i,\ \forall\, i \in [0,k)$ are *balanced*, i.e., $|H_i| \leq (1+\varepsilon)\left\lceil |O|/k \right\rceil$.

A *graph* G is an ordered pair (V, E), where V is a set of elements called *vertices* and $E \subseteq V \times V$ is a set of *edges*.

Definition 3.2.1 (Nested Bipartition)**.** *Let $G = (V, E)$ be a graph and $\ell \in \mathbb{N}^+$.*

*A **nested bipartition** $\mathcal{H}^2_\ell(G)$ of G into ℓ levels is a sequence $\left(\mathcal{H}_i(V)\right)_{i=0}^{\ell-1}$, where each $\mathcal{H}_i(V)$ is a $2^{\ell-i}$-partition of V, i.e.,*

$$\mathcal{H}_i(V) = \{H^i_0, H^i_1, \dots, H^i_{2^{\ell-i}-1}\}.$$

These partitions satisfy the nesting property:

$$H^i_j = H^{i-1}_{2j} \cup H^{i-1}_{2j+1}, \quad \text{for all } i \in [1,\ell) \text{ and } j \in [0, 2^{\ell-i}).$$

A nested bipartition can also be visualized as a tree, i.e., a connected graph without cycles, where the vertices of the tree are sets of vertices and edges represent set inclusion. This means that the root of the tree is $\mathcal{H}_\ell$, and its two children are the elements of $\mathcal{H}_{\ell-1}$. For

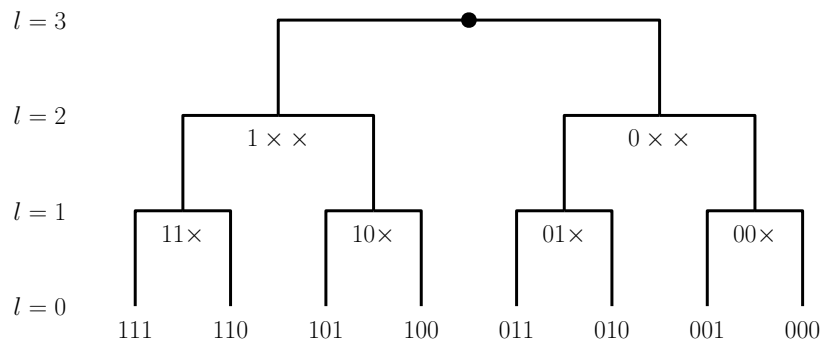


Figure 3.1: Illustrated is a nested bipartition with 3 levels, and the cell IDs with 3 bits per leaf vertex. Consider a vertex $u \in V$ in the cell $H_i \in \mathcal{H}_0$, with $i_{(2)} = 011_{(2)}$. Then, $c_{\mathrm{ID}}(u, 2)$ can be computed by removing the first two least significant bits, i.e., $c_{\mathrm{ID}}(u, 2) = 0\text{XX}_{(2)} = 0_{(2)}$.

each of these two children $H' \in \mathcal{H}_{\ell-1}$, the two children are the elements of $\mathcal{H}^2(H')$. This continues until all the elements of $\mathcal{H}_0$ are leaves in the tree.

We say an edge $e = (u, v) \in E$ is a *cut edge* of level l if both endpoints lie in different cells on level l. An important feature of a *nested* bipartition is based on the fact that a cut edge on level l remains a cut edge on all lower levels $j,\ 0 \leq j \leq l$.

Additionally, for two vertices $u, v \in V$, we define the *lowest common level* $\mathrm{LCL}(u, v)$ as the level i with $0 \leq i \leq \ell$ such that u and v lie in the same cell on level i, but in different cells on all lower levels than i. Note that if u and v do not share any cells, the LCL is ℓ.

Given $\mathcal{H}^2_\ell(G)$, we define the *cell ID* of a vertex $u \in V$ on level $l \in [0, \ell)$ as $c_{\mathrm{ID}}(u, l)$, with $c_{\mathrm{ID}} : V \times [0, \ell) \mapsto \{0, 1\}^\ell$, such that

$$c_{\mathrm{ID}}(u, l) = i_{(2)} \iff u \in H_i \in \mathcal{H}_l,$$

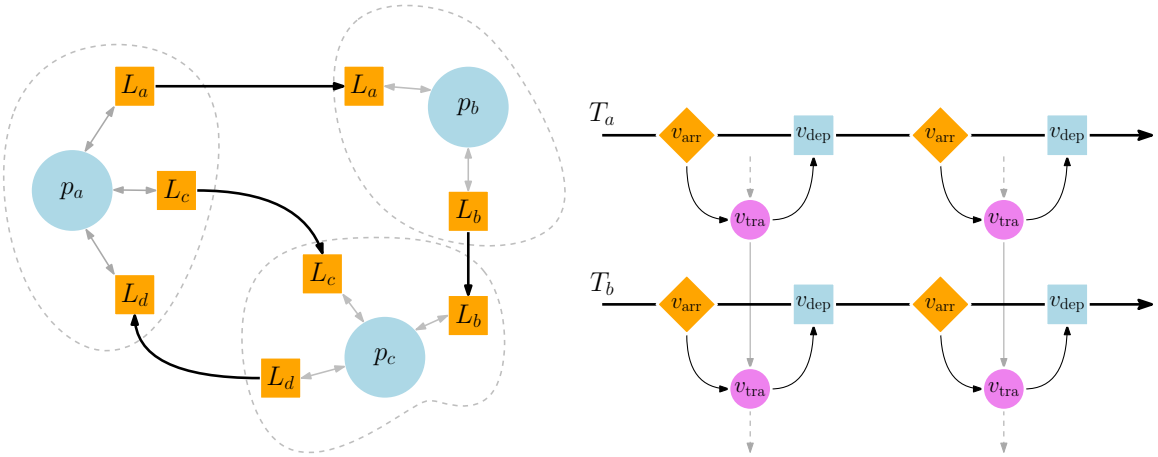
where $i_{(2)} \in \{0, 1\}^\ell$ denotes the binary representation of the integer i. To simplify the notation, we write $c_{\mathrm{ID}}(u)$ for $c_{\mathrm{ID}}(u, 0)$. Note that given $c_{\mathrm{ID}}(u) = i_{(2)}$, the cell ID $c_{\mathrm{ID}}(u, l)$ can be computed by removing the l least significant bits from $i_{(2)}$, i.e.,

$$c_{\mathrm{ID}}(u, l) = \left\lfloor \frac{c_{\mathrm{ID}}(u)}{2^l} \right\rfloor.$$

This can be seen from the representation as a tree, cf. Figure 3.1.

3.3 Modeling as Graph

Finding shortest paths on graphs is one of the fundamental problems in computer science, and there are several polynomial algorithms that solve this problem on graphs with non-negative edge weights. Probably the best known algorithm is Dijkstra's algorithm [14],



(a) The time-dependent graph G^{δ}.

(b) The time-expanded graph G^{ε}.

Figure 3.2: The left shows the *time-dependent* graph G^{δ}; the right the *time-expanded* G^{ε}. In the time-dependent model, a *stop vertex* is represented by a blue circle and a *line vertex* by orange squares. The gray arrows model transfers towards lines, and the black, thicker arrows model the time-dependent edges. In order to illustrate which vertices belong to a stop, stops are surrounded with a dashed border. In the time-expanded model, the three types of vertices are differentiated with different shapes and colors: *transfer vertices* are pink round circles, *arrival vertices* are orange diamonds and *departure vertices* are blue squares.

which solves the problem in $\mathcal{O}(m + n \log n)$. In order to compute optimal journeys through a public transit timetable using Dijkstra's algorithm, it is necessary to determine the structure of the query graph, i.e., how vertices, edges, and, if applicable, edge weights are defined. In the following, we will look at two well-known approaches, the *time-dependent* graph $G^{\delta} = \left(V^{\delta}, E^{\delta}\right)$ and *time-expanded* graph $G^{\varepsilon} = (V^{\varepsilon}, E^{\varepsilon})$.

3.3.1 Time-Dependent Graph

The time-dependent graph $G^{\delta} = \left(V^{\delta}, E^{\delta}\right)$ models stops as vertices and edges between them hold piecewise linear functions, which map departure time to trip duration. This results in a compact graph where edge weights no longer have scalar values, so algorithms like TDD [13] (Time-Dependent Dijkstra) need to evaluate time functions when relaxing edges. To compute Pareto optimal solutions, stop vertices are modeled in more detail by adding a *line vertex* for each line $L \in \mathcal{L}$ departing from the corresponding stop $p \in \mathcal{S}$. All line vertices of p are connected to the stop vertex of p via two directed edges (one forward, one backward edge), which represents hopping and off a line. The direction from line vertex to stop vertex has a transfer cost of one, the return direction zero. These are called *entering transfers* as well as *exiting transfers*. Along a line $L \in \mathcal{L}$, all corresponding line vertices of

all stops $p \in \Pi(L)$ are connected in the direction of travel. See [13] for more details; in this thesis, we use the graph using additional line vertices, as presented in Figure 3.2a.

Based on a time-dependent graph model, Timetable Labeling (TTL) [23] computes extra labels per stop to answer queries within a few microseconds. Due to the nature of the graph model, the technique cannot handle footpaths between stops, and cannot compute Pareto-optimal solutions.

Geisberger [24] presents an adapted variant of CH [25] (Contraction Hierarchies), based on a slightly different time-dependent graph model. Contraction Hierarchies [25], a very successful technique for shortest paths on road networks, iteratively removes "unimportant" vertices and inserts shortcuts between neighboring vertices to preserve shortest-path distances. As Geisberger [24] shows, CH benefits from good hierarchies in road networks, whereas public transportation networks have limited hierarchies. Furthermore, algorithms for time-dependent networks suffer from the fact that the arrival time at the destination is not known and therefore no additional backward search is possible; unlike road graphs. This is also reflected in the performance of queries: On all networks, the CH variant achieves query times of less than $1\,\mathrm{ms}$ (these correspond to speedups between six and 43), but the networks are relatively small (compared to the data sets used by e.g., [3, 5, 10]). Geisberger [24] predicts that incorporating and modeling footpaths would be detrimental to the CH approach.

3.3.2 Time-Expanded Graph

The time-expanded model, on the other hand, builds the time dimension into its graph topology $G^{\varepsilon} = (V^{\varepsilon}, E^{\varepsilon})$ by modeling elementary events that happen at a certain time. See Figure 3.2b as an illustration and a comparison between the two different graph models. An event $\varepsilon \in \mathcal{E}$ is represented in this graph as three vertices, an *arrival* $v_{\mathrm{arr}}(\varepsilon)$, *departure* $v_{\mathrm{dep}}(\varepsilon)$ and *transfer* vertex $v_{\mathrm{tra}}(\varepsilon)$. A trip $T \in \mathcal{T}$ alternately connects departure and arrival vertices along its stops $\Pi(T)$ via edges, i.e.,

$$\forall\, i \in [1, |T|) : (v_{\mathrm{dep}}(T[i]), v_{\mathrm{arr}}(T[i+1])) \in E^{\varepsilon} \text{ and}$$
$$\forall\, i \in (1, |T|) : (v_{\mathrm{arr}}(T[i]), v_{\mathrm{dep}}(T[i])) \in E^{\varepsilon}.$$

Each arrival vertex is also connected to its corresponding transfer vertex. To enable transfers between trips, all transfer vertices of a stop are connected linearly, i.e., in chronological order, with an edge. Compared to the time-dependent graph, the graph is significantly larger. Edge weights are now scalar again, e.g., one can use the time difference between two vertices as edge weights. Since the time information of the network is contained in the vertices themselves, an *earliest arrival* query is merely a reachability query on the time-expanded graph. Starting from the first reachable departure event at the source stop, an algorithm only needs to find the first arrival event at the destination stop. This can be done using TED [12] (Time-Expanded Dijkstra), which solves the *earliest arrival* problem on the time-expanded graph.

Public Transit Labeling

Note that due to the implicit time representation in the time-expanded graph, only *reachability* between two events must be tested during the query. Given a blackbox algorithm that, for any two vertices $u, v \in V$, returns whether a path exists from u to v, the time-expanded graph can be used to solve an earliest arrival query efficiently. The procedure is as follows: the first reachable departure event is selected as u, and then one iterates over all arrival events at the target stop to find the first arrival event v such that a path between u and v exists. This is essentially the idea of PTL [16] (Public Transit Labeling), and the reachability blackbox algorithm is Hub Labeling (HL). HL [17] assigns two sets of vertices, referred to as *hubs*, to each vertex $u \in V$: a forward label $L_f(u)$ and a backward label $L_b(u)$. The forward label $L_f(u)$ stores, along with each hub w, the shortest-path distance from u to that w. Analogously, the backward label $L_b(u)$ stores the distances from each hub to u. For correctness, all labels must fulfill the *cover property*, which states that for each pair of vertices $u, v \in V$ the intersection of $L_f(u) \cap L_b(v)$ contains at least one vertex w on the shortest path from u to v (if such a path exists). The distance between u and v is then given by the minimum sum of the forward and backward distances over all hubs in the intersection. In PTL, Hub Labeling is only used for reachability queries between vertices of the time-expanded graph, only the hubs are stored in the labels; the distances are not needed.

3.4 Transfer Patterns

Transfer Pattern [6] is a journey planning algorithm that uses precomputed information during the query to answer queries very quickly and is the building block of Google Maps transit routing [26]. The basic idea of Transfer Patterns is based on the fact that for a source and a target stop across all possible departure times, many optimal journeys share lines, and even stops where one has to change trains. These "commonalities" are exploited by extracting the lines and transfer stops used from each possible optimal journey into a common graph, which can be seen as a time-independent abstraction of all optimal journeys. This graph has stops as vertices, and edges are directed and represent the lines between the two corresponding stops. To distinguish the *type* of stop in the graph, there are 3 types of vertices. The *root* vertex, i.e., the source stop, a *target* vertex for each reachable (target-)stop and a *prefix* vertex for each transfer stop. Note that a stop can occur any number of times as a prefix vertex for different target stops. Due to the time flow implicitly represented in this graph, the transfer patterns for any source stop is a directed acyclic graph (DAG). For a source stop p_s and a target stop p_t, we denote the transfer patterns as $\mathcal{TP}(p_s, p_t)$, which is a set of stop sequences. See Figure 3.3 as an example.

The respective graph must be saved for every pair of source and target stops, which results in enormous memory consumption. For a stop p_s, common parts of all transfer patterns $\mathcal{TP}(p_s, \cdot)$ can be assembled in advance. Then, at query time, the correct DAG corresponding to the destination stop p_t must be extracted. This is why the edges are also directed

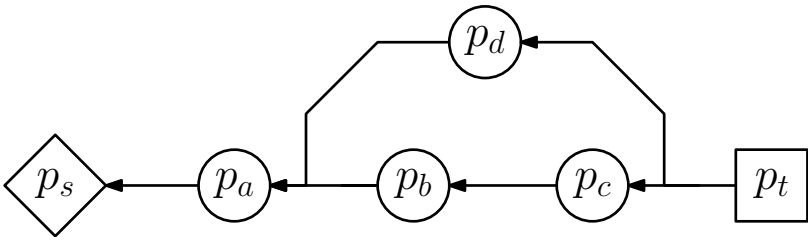


Figure 3.3: An example transfer patterns $\mathcal{TP}(p_s, p_t)$ using the same notation as in the original paper [6], i.e., the root vertex is a diamond, the target vertices a square and prefix vertices are circles. Here, the optimal transfer patterns from p_s to p_t are the sequences $\langle p_s, p_a, p_b, p_c, p_t \rangle$ and $\langle p_s, p_a, p_d, p_t \rangle$.

backwards, as all edges are first extracted starting from the target. A time-dependent multicriteria variant of Dijkstra's algorithm is run on this graph, which for each edge it relaxes, extracts the fastest possible reachable connection from a precomputed lookup-index. The query is very fast, which is due to the fact that the search space is very small and the lookup-index allows for fast access of time information. However, the precomputation must answer every possible query in advance and therefore does not scale well at all.

To reduce memory consumption, stops that occur in many transfer patterns (and are therefore in many optimal journeys) are marked as *hubs*. The $\mathcal{TP}$ computation stays same for hubs, i.e., all the transfer patterns to all other stops are computed, but for all non-hub stops, the precomputation must be adapted slightly. For a non-hub stop p_s, only *local* searches are started which either end in other non-hub stops or which lead via hubs. Then only the transfer patterns up to hubs stations are stored in $\mathcal{TP}(p_s)$. This means that if p^* is a hub and $\langle p_s, p_a, p^*, p_b, p_t \rangle$ is an optimal transfer patterns for the source stop p_s, then only $\langle p_s, p_a, p^* \rangle$ is inserted into the $\mathcal{TP}(p_s)$. During a query for p_s to p_t, the transfer patterns of all reachable hubs $\mathcal{X}$ and of the source are merged, i.e., the query graph is build using $\mathcal{TP}(p_s, p_t), \mathcal{TP}(p_s, x)$ and $\mathcal{TP}(x, p_t), \ \forall x \in \mathcal{X}$. Afterward, the query algorithm stays the same as without the use of hubs. However, the hub approach only reduces the memory consumption, not the precomputation. This is why the authors introduce heuristics, e.g., to limit the search from non-hubs to hubs to a maximum of two transfers. In rare cases, this leads to suboptimal solutions for some queries, but speeds up the precomputation.

3.4.1 Scalable Transfer Patterns

One approach to tackle the poor precomputation time and memory consumption is Scalable Transfer Patterns [18]. The idea is to exploit the hierarchy of long-distance and local traffic by distinguishing between local and global transfer patterns. To do this, the algorithm first computes a partition $\mathcal{H}(\mathcal{S})$ of all the stops, and then transfer patterns *inside* and *outside* of each cell $H \in \mathcal{H}(\mathcal{S})$ are calculated. Stops, from which connections to stations outside the current cell H depart, are *borderstops* of H, which we denote as $\bar{H}$. Within each cell

H, all *local* transfer patterns $\mathcal{TP}^{\downarrow}$ between all station pairs of H are computed. To enable routing between clusters, *global* transfer patterns $\mathcal{TP}^{\uparrow}$ are computed between borderstops. Bast et al. introduce the notion of *Convex Transit Cluster* (see [18], Definition 1), which guarantees that all queries within this cluster $H \in \mathcal{H}(\mathcal{S})$, i.e., $p_s, p_t \in H$, only run via stops of this cluster. A query from p_s to p_t must therefore check whether both stops are in the same convex cluster. If this is the case, it is sufficient to look at $\mathcal{TP}^{\downarrow}(p_s, p_t)$ and, analogous to the original transfer patterns query, build the query graph from it. If both stops are either in the same non-convex cells or in different cells $H_i, H_j \in \mathcal{H}(\mathcal{S})$, such that $p_s \in H_i, p_t \in H_j$, the following transfer patterns must be merged: $\mathcal{TP}^{\downarrow}(p_s, i), \mathcal{TP}^{\uparrow}(i, j)$ and $\mathcal{TP}^{\downarrow}(j, p_s), \forall\, i \in \bar{H}_i,\ j \in \bar{H}_j$.

3.5 Connection Scan Algorithm

CSA [10] (Connection Scan Algorithm) is a simple, effective routing algorithm based on the insight that the time-expanded graph is a DAG. Instead of a graph data structure, all trip edges of the time-expanded graph (the connections) are stored in topological order in an array $\mathcal{C}$. This makes the precomputation of CSA very fast; only the array $\mathcal{C}$ needs to be sorted. In contrast to the time-expanded graph, footpaths are not incorporated as multiple edges between events (of the corresponding stops); instead, CSA maintains an additional footpath graph with stops as vertices.

In the following, we describe the basic variant of CSA in more detail so that the basic idea of the algorithm, which is the building block of some variants, becomes clear. See Algorithm 1.

Given a fixed departure time query $q = (p_s, p_t, \tau)$, all connections $c \in \mathcal{C}$ with $\tau_{\text{dep}}(c) \geq \tau$ are scanned in sorted order. CSA maintains the tentative arrival time $\mathbb{S} : \mathcal{S} \mapsto \tau$ for each stop, and for each trip, a bit $\mathbb{T} : \mathcal{T} \mapsto \{\text{False}, \text{True}\}$ indicating whether the trip has been boarded. The algorithms starts by finding the first connection $c^0 \in \mathcal{C}$ which departs after τ using a binary search. Starting with c^0, each connection $c \in \mathcal{C}$ is then scanned to check whether it can be from p_s at τ. This is the case if the corresponding trip has already been boarded (i.e., the bit of $T(c)$ is set) or if the departure stop $p_{\text{dep}}(c)$ has already been reached by other means (i.e., the tentative arrival time is less than or equal to the departure time $\tau_{\text{dep}}(c)$). If so, the connection is scanned and the arrival time at p_t may be improved. Whenever an improvement is made, all outgoing footpaths $\mathcal{F}^{\uparrow}(p_{\text{arr}}(c))$ are *relaxed*[1]. The scanning process that stop as soon as a connection c' is processed, which departs later than the tentative arrival time at the target stop p_t.

Unfortunately, the basic version of CSA only computes the earliest possible arrival time; for Pareto-optimal solutions, CSA must solve the more difficult profile problem. For the earliest arrival profile problem, each stop is assigned a profile (a list of tuples of departure and arrival times) instead of an arrival time, and the connections are scanned backwards

[1] In graph algorithm jargon, "relaxing" an edge (u, v) means trying to update a metric at v by traversing that edge. In our case, the metric is the arrival time.

Algorithm 1: Earliest arrival CSA.

Procedure `Query`$(p_s, p_t, \tau_{\text{dep}})$
  $\mathbb{S}(p) \leftarrow \infty,\ \forall p \in \mathcal{S}$ `// arrival times`
  $\mathbb{T}(T) \leftarrow$ False, $\forall T \in \mathcal{T}$ `// reachability bits`
  $\mathbb{S}(p_s) \leftarrow \tau_{\text{dep}}$
  for each $p \in \mathcal{F}^{\uparrow}(p_s)$ **do** `// relax transfers`
    $\mathbb{S}(p) \leftarrow \tau_{\text{dep}} + \Delta\tau_{\text{fp}}(p_s, p)$
  $c^0 \leftarrow$ `FirstReachConn`(p_s, τ_{dep}) `// binary search`
  for each $c^0 \preceq c \in \mathcal{C}$ (sorted by dep. time) **do**
    if $\mathbb{S}(p_t) \leq \tau_{\text{dep}}(c)$ **then break** `// target pruning`
    if $\mathbb{T}(T(c)) =$ True $\vee\ \mathbb{S}(p_{\text{dep}}(c)) \leq \tau_{\text{dep}}(c)$ **then** `// reachable`
      $\mathbb{T}(T(c)) \leftarrow$ True
      if $\tau_{\text{dep}}(c) < \mathbb{S}(p_{\text{dep}}(c))$ **then** `// update arr. time`
        $\tau_{\text{dep}}(c) \leftarrow \mathbb{S}(p_{\text{dep}}(c))$
        for each $p \in \mathcal{F}^{\uparrow}(p_{\text{dep}}(c))$ **do** `// relax transfers`
          $\mathbb{S}(p) \leftarrow \min\{\mathbb{S}(p), \tau_{\text{dep}}(c) + \Delta\tau_{\text{fp}}(p_{\text{dep}}(c), p)\}$

in time. For each connection, it is checked whether this leads to an improvement in the profile of the departure stop. For Pareto-optimality, the arrival times in the profiles are now replaced by fixed sized vectors, with the i-th position representing the arrival time with i trips. Maintaining the profiles takes a lot of time (especially compared to simply updating one arrival time). Instead of naively maintain a list of vectors, one can improve the performance by leveraging SIMD intrinsics[2]. However, this vectorized approach imposes an upper limit of 7 transfers with 256-bit SSE registers (or up to 15 with 512-bit AVX2). Nonetheless, the runtime is significantly worse than the basic variant, so the authors drop the Pareto criterion and optimize the number of transfers as a second criterion (the first still being arrival time). This variant has hardly any overhead compared to the earliest arrival profile variant, as the number of transfers is cleverly encoded in the bits of the arrival time, meaning that the algorithm barely needs to be changed.

One problem with the Pareto optimal profile variant of CSA is that the "end" of the profile is unknown, i.e., it is unclear what the latest possible arrival time with minimal transfers is. Therefore, in order to remain accurate, the profile variant of CSA must always start from the last connection (and thus in the worst case always scan all connections). For the non-Pareto profile problem, on the other hand, a normal CSA query can be executed from τ_{end} to efficiently compute the latest possible arrival time. The authors therefore propose the following heuristic to efficiently answer profile queries in practice: The latest possible arrival time τ_{arr} is computed, and the time horizon is bounded by $\tau_{\text{end}} = \tau_{\text{start}} + 2(\tau_{\text{arr}} - \tau_{\text{start}})$.

[2]SIMD (Single Instruction, Multiple Data) refers to a class of CPU instructions that perform the same operation on multiple data elements simultaneously. For more information, see `https://en.algorithmica.org/hpc/simd/`.

Even though it only covers part of the profile, the *range problem* still produces useful and realistic results in practice.

All in all, CSA benefits significantly from the high cache locality of sorted connections and the use of simple data structures, particularly for the earliest arrival problem. However, this straightforward, cache-friendly design carries an inherent drawback: if a connection that is unnecessary, e.g., because it is unreachable, resides in the cache, it makes little difference whether it is ignored or processed, since processing a single connection is relatively lightweight.

3.5.1 Partitioning the Connections

To tackle this "filtering" problem, the following high-level scheme was introduced with ACSA [10] (Accelerated Connection Scan Algorithm).

(i) **Partitioning**: Instead of a single large connections array, all connections $\mathcal{C}$ are partitioned into many smaller sorted subarrays $(\mathcal{C}_1, \mathcal{C}_2, \dots)$.

(ii) **Selection**: At query time, identify which subarrays $\mathcal{C}_i$, $i \in I$ contain connections relevant to answer the p_s–p_t query.

(iii) **Merge**: Efficiently merge the selected subarrays into one temporally sorted array $\mathcal{C}_{\text{query}} = \bigcup_{i \in I} \mathcal{C}_i$.

(iv) **Scan**: Execute the standard CSA using $\mathcal{C}_{\text{query}}$ to compute optimal journeys.

Here is an intuitive example of how this scheme works: Consider a query from Karlsruhe to Munich. Scanning the entire German timetable, including e.g., connections in Berlin, could waste time. Instead, suppose that during preprocessing we assign

- all local Karlsruhe connections to one subarray $\mathcal{C}_{\text{KA}}$,
- all local Munich connections to another subarray $\mathcal{C}_{\text{MU}}$, and
- all long-distance connections to a third subarray $\mathcal{C}_{\text{long-dist}}$.

At query time, we need only merge those three subarrays into one temporally sorted array $\mathcal{C}_{\text{query}} = \mathcal{C}_{\text{KA}} \cup \mathcal{C}_{\text{MU}} \cup \mathcal{C}_{\text{long-dist}}$, and then run CSA using $\mathcal{C}_{\text{query}}$.

Of course, simply grouping by geography or agency risks pruning optimal journeys through other cities, so the critical part of ACSA is determining which connections belong in which subarray in order to preserve optimality.

ACSA achieves this by first partitioning the set of stop into k cells across l levels. For each level i, $1 \leq i \leq l$, and all the k^i cells on this level, the connections are stored in an array $\mathcal{C}^i_j$, $1 \leq j \leq k^i$, which are required to travel through the current cell optimally. Note that $\mathcal{C} = \bigcup_{1 \leq i \leq l} \bigcup_{1 \leq j \leq k^i} \mathcal{C}^i_j$.

For the query, only the arrays of the cells that are actually "important" are merged. To reduce the number of merge operations and to keep the memory consumption linear, each connection $c \in \mathcal{C}$ is only stored in the highest possible level i', instead of storing it for

each level $1 \leq i \leq i'$. To compute which connections belong to $\mathcal{C}_j^i$ for cell j on level i, ACSA performs a profile query for each *incoming* connection and extract all optimal journeys towards all *outgoing* connections. To speed up computation, ACSA works in a bottom-up fashion and for each level $i > 1$ only uses connections of $\bigcup_{1 \leq j \leq k^{i-1}} \mathcal{C}_j^{i-1}$. One disadvantage of storing all connections $\mathcal{C}$ in many small arrays $(\mathcal{C}_1, \mathcal{C}_2, \dots)$ arises during the merging phase in the query. As the authors of ACSA [10] report, the query is dominated by computing $\mathcal{C}_{\text{query}}$; pure scanning of connections is very cheap, as in the basic CSA. This has a particular impact on metropolitan regions, such as London or Paris, the authors report a slowdown of ACSA compared to CSA. We confirm these findings in our own experiments (cf. Section 5). Nevertheless, ACSA showed that multilevel approaches could still be competitive on public transit, albeit not solving the Pareto-optimal problem.

Another algorithm, which is based on the scheme mentioned above of maintaining multiple subarrays $(\mathcal{C}_1, \mathcal{C}_2, \dots)$, is GDCSA [27] (Goal-Directed CSA). The set of stops is divided into geographical regions $\mathcal{H}$, and between each pair of regions $(\mathcal{H}_A, \mathcal{H}_B) \in \mathcal{H} \times \mathcal{H}$, lower and upper bounds are precomputed with respect to the travel time from $\mathcal{H}_A$ to $\mathcal{H}_B$. These bounds are evaluated during the query in order to merge only the connection arrays of regions that are reachable. The authors evaluate GDCSA only for multi-criteria Pareto problems, namely on the 4 criteria (maximizing) departure time, (minimizing) arrival time, (minimizing) the number of transfers and (minimizing) the total walking distance. Compared to the 4-criteria Pareto variant of CSA, GDCSA achieves a speedup of 2.5 to 9, but the authors do not evaluate their algorithm for Pareto-optimal queries with respect to (minimizing) arrival time and (minimizing) the number of transfers. Since this is a simpler problem, it is unclear whether GDCSA would achieve a similar speedup (especially in regard to the fact that merging the arrays dominates the runtime for simpler problems; see ACSA above).

3.6 Trip-Based Public Transit Routing

TB [1] (Trip-Based Public Transit Routing) is another Pareto-optimal journey planning algorithm and works like RAPTOR in rounds through the network. However, it does not scan lines, but trips. A set of transfers $\mathfrak{T} \subseteq \mathcal{E} \times \mathcal{E}$ is precomputed in order to directly find the earliest trip that can be reached when relaxing a transfer. This transfer generation is very fast and easy to parallelize, and only takes minutes on large networks.

3.6.1 Transfer Precomputation

This section describes how and which transfers are precomputed.

Only *valid* transfers are generated, i.e., for all transfers $\mathfrak{t} = (T_a[j], T_b[i])$ a change between the events must be possible in terms of time

$$\tau_{\text{arr}}\left(T_a[j]\right) + \Delta\tau_{\text{fp}}(p(T_a[j]), p(T_b[i])) \leq \tau_{\text{dep}}\left(T_b[i]\right).$$

Algorithm 2: TB U-turn Reduction. Taken from [1].

```
1 Procedure Reduce(𝔗)
2     for each 𝔱 ∈ 𝔗 do
3         (T_a[j], T_b[i]) ← 𝔱
4         p ← p(T_a[j − 1])
5         q ← p(T_b[i + 1])
6         valid ← τ_arr(T_a[j − 1]) ≤ τ_dep(T_b[i + 1])
7         if p = q ∧ valid then
8             𝔗 ← 𝔗 \ {𝔱}
```

For each trip $T \in \mathcal{T}$ and each event $T[j]$, $j \in [2, |T|]$, all reachable lines $\mathcal{L}_{\text{rea}}(p(T[j]))$ that can be reached via a footpath (or no footpath) are collected. Then, for each line $L \in \mathcal{L}_{\text{rea}}$, the earliest trip T_b of L is used to generate a valid transfer $\mathfrak{t} = (T_a[j]T_b[i])$, $i \in [1, |T_b|)$. However, this approach leads to many superfluous transfers, most of them are not necessary to answer queries optimally. Therefore, a set of pruning rules was proposed in [1] to reduce the number of resulting transfers without discarding transfers required to answer queries optimally.

Transfers between trips of the same line are pruned if staying seated in T_a is preferred, i.e., if $T_a \preceq T_b$ and $j \leq i$. If we consider a transfer $\mathfrak{t} = (T_a[i], T_b[i])$ with $p(T_a[j-1]) = p(T_b[i+1])$ and $\tau_{\text{arr}}(T_a[j-1]) \leq \tau_{\text{arr}}(T_b[i+1])$, then $\mathfrak{t}$ is not required. This is known as *U-turn* rule; see Algorithm 2.

The last rule, which is referred to as *latest-exit* rule by [5], removes transfers which are "dominated" by others starting from the same trip. For a transfer $\mathfrak{t} = (T_a[j], T_b[i])$, consider the journey $J = \langle T_a[k, j], \mathfrak{t}, T_b[i, l] \rangle$. The transfer can be pruned if there exists another journey $J' = \langle T_a[k, j'], \mathfrak{t}', T_c[i', l'], f \rangle$, with $j < j'$ and $f = (p(T_c[l']), p(T_b[l]))$ which has a better (or equal) arrival time at $p(T_b[l])$. There is a trade-off between reducing the set of transfers by identifying dominating journeys for J and maintaining speed. Since the transfer generation phase should be fast, many potentially dominating journeys, especially those that do not start with the same trip as J, are not considered. See [5] for a different transfer generation given a transitive footpath model.

In [1], all valid transfers are first generated for each trip and then reduced, this can take a lot of time; especially if many transfers are removed again. An alternative transfer generation approach has therefore been proposed by [28], which improves preprocessing time by discarding a larger number of superfluous transfers early, before they are generated.

The entire transfer generation can be easily parallelized, as trips can be processed independently.

Algorithm 3: TB trip scanning operation (for target stop p_t). Taken from [5].

```
1  Procedure Scan(Q_n, τ_min)
2      Q_{n+1} ← ∅
3      for each T[j,k] ∈ Q_n do
4          for i from j to k do
5              if τ_arr(T[i]) ≥ τ_min then break                    // target pruning
6              if τ_arr(T[i]) + Δτ_fp(p(T[i]), p_t) < τ_min then    // create target label
7                  τ_min ← τ_arr(T[i]) + Δτ_fp(p(T[i]), p_t)
8                  L ← L ∪ {(τ_min, n)}, removing dominated labels
9      for each T[j,k] ∈ Q_n do
10         for i from j to k do
11             if τ_arr(T[i]) ≥ τ_min then break                    // target pruning
12             for each (T[i], T'[i']) ∈ 𝔗 do                       // relax transfers
13                 t ← (T[i], T'[i'])
14                 Enqueue(t, Q_{n+1})
```

Algorithm 4: TB enqueuing operation. Taken from [5].

```
1  Procedure Enqueue((T_a[i], T_b[j]), Q_n)
2      if R(T_b) ≤ j then return                  // trip segment already reached
3      Q_n ← Q_n ∪ {T_b[j, R(T_b) − 1]}
4      for each T' ⪰ T_b do                       // update reached index
5          if R(T'_b) ≤ j then break
6          R(T'_b) ← j
```

3.6.2 Fixed Departure Time Query Algorithm

Conceptually, the core idea behind the TB query is similar to that of RAPTOR, in that the network is explored in a breadth-first manner. However, rather than scanning lines, TB processes individual trip segments in each round. Thanks to precomputed transfers, TB can access the earliest reachable trip on a line directly; whereas RAPTOR needs to invest additional time to find it.

A key advantage of expanding the search in discrete rounds, with round $i \in \mathbb{N}$ exploring all journeys using exactly i trips, is that it enables the efficient, implicit representation of Pareto-optimal solutions. In practical applications, users only consider journeys where the number of transfers is usually small and bounded, often by a constant $k = 8, 16, \ldots$. This boundedness makes it feasible to allocate a fixed-size array for each stop, where the i-th entry stores the best-known arrival time using exactly i trips. The overhead associated with dynamically maintaining Pareto sets is hence avoided. Dynamic Pareto sets require complex operations, such as inserting new labels, checking for dominance and potentially

removing dominated entries, all of which involve non-trivial memory management, including reallocations and indirections. These operations increase runtime and lead to poorer cache performance due to scattered memory access patterns. By contrast, the fixed-size, "per-round" representation significantly reduces this overhead. Updates and lookups can be performed in constant time using random access. Furthermore, optimal journeys do not need to be stored explicitly; they are encoded implicitly via parent pointers that reference the previous trip segment used to reach a certain event. This compact representation simplifies data structures and reconstruction of optimal journeys.

To keep track of which parts of the network have already been explored, the TB query algorithm maintains a *reached index* $R : \mathcal{T} \mapsto \mathbb{N}^+$. For each trip $T \in \mathcal{T}$, the reached index $R(T)$ stores the smallest stop index $i \in [1, |T|]$ such that the suffix $T[i, |T|)$ has already been scanned. In addition, the algorithm works on several FIFO queues Q_i, $i = 1, 2, \ldots, k$, which contain the trip segments to be scanned per round i. For the target stop, a set of *target labels* $\mathcal{L}$ is also maintained, which encode the Pareto-optimal journeys. Pseudocode of the main algorithm is given in Algorithm 3, which shows the scanning process in round n.

The Scan algorithm consists of two loops, where the first loop is responsible for creating new solutions, and the second for relaxing transfers and filling the next queue Q_{n+1}.

In the first loop, the algorithm iterates over all trip segments $T[j, k]$ in the current queue Q_n. For each event $T[i]$, where $j \leq i \leq k$, an inner loop (Lines 4–8) checks whether the arrival time at the target stop can be improved. If the arrival time of the event already exceeds the best known arrival time τ_{min} at the target (Line 5), the trip segment can be skipped from that point onward. This is because all subsequent events $T[l]$, for $l \geq i$, will have even later arrival times. This optimization is known as *target pruning*. Otherwise, the algorithm checks whether the generated solution using the current event improves τ_{min} (Line 6); if yes, $\mathcal{L}$ is updated using the newly found solution.

The second loop also iterates over all trip segments $T[j, k]$ of Q_n. For each trip segment, the inner loop (Lines 10–14) relaxes the outgoing transfers of events $T[i]$, $j \leq i \leq k$. For each such transfer $\mathfrak{t} = (T[i], T'[i']) \in \mathfrak{T}$, the algorithm invokes the Enqueue method to insert any newly reachable trip segments into the next queue.

As in the first loop, target pruning is applied to avoid processing segments that cannot lead to an improved arrival time. The Enqueue operation is implemented efficiently using the reached index data structure, as described in Algorithm 4. The decision to split the work into two separate loops – one for computing new arrival times and one for processing transfers – is motivated by performance optimization for modern hardware and by a stronger target pruning. Processing arrival times and transfers in isolation improves cache locality and prevents unrelated data from polluting the cache simultaneously. This separation allows for more efficient memory access and results in noticeable speed improvements in practice. To fill the first queue Q_1, the query algorithm collects all reachable lines, i.e., all lines, which depart at p_s or any footpath-reachable stop. For each of these lines, the algorithm finds the earliest reachable trip and enqueues the trip segment analogous to the Enqueue operation.

Algorithm 5: Profile Query: High-level overview.

```
1 Procedure ProfileQuery(q = (p_s, p_t, [τ_start, τ_end]))
2     J ← ∅
3     D ← all departures reachable from p_s during [τ_start, τ_end]
4     D ← sort descending by departure time
5     for each ε = T[i] ∈ D do
6         run query for q = (p_s, p_t, τ_dep(ε) + Δτ_fp(p_s, p(ε)))
7         J_run ← newly found journeys
8         J ← J ∪ J_run
```

Further optimizations are, for example, not to maintain k queues, but a long queue Q^*. As each event can be used to identify at most one trip segment, Q^* can be preallocated with a length of $|\mathcal{E}|$. For more optimizations, we refer to the original paper [1].

3.6.3 Journey Unpacking

As previously mentioned, optimal journeys are not stored explicitly. Instead, they are represented implicitly using parent pointers. This compact representation avoids the need to store full journey descriptions during the algorithm's execution. For each trip segment $T_n[i_n, j_n]$ scanned during round n, a pointer to the corresponding preceding trip segment $T_{n-1}[i_{n-1}, k_{n-1}] \in Q_{n+1}$ is stored. This pointer identifies the segment from the previous round that led to the current one via a transfer. In addition, the algorithm stores the position $j_{n-1} \in [i_{n-1}, k_{n-1}]$ that references the specific event $T_{n-1}[j_{n-1}]$ from which the transfer $(T_{n-1}[j_{n-1}], T_n[i_n])$ originated. Every label in $\mathcal{L}$ keeps track of the trip segment that led to the journey. Using this information, a journey can be reconstructed by tracing back the chain of parent pointers, starting from the final trip segment and iteratively following the pointers all the way back to the first round Q_1.

3.6.4 Profile Query Algorithm

Due to the third criterion (maximizing departure time), profile queries allow an additional pruning rule when enumerating journeys from late to early departure. The main insight is that journeys that have already been found can be used for pruning, since for two journeys with the same arrival time and number of transfers, the later departure journey is preferred.

Therefore, the profile query works as follows (both for TB and for RAPTOR), see Algorithm 5. First, all departures that depart during $[\tau_{\text{start}}, \tau_{\text{end}}]$ and can be reached from the source stop are collected and sorted in descending order by departure time. Now, a fixed departure time query is run for each departure time; but after this run, the data structures (e.g., arrival times per stop per round) are not reset. Since newly found journeys will over-

write old journeys, one has to extract all journeys after each run. In addition, the reached index must be slightly changed to store the earliest index of a trip for each number of trips. See [1, 5] for more details.

3.6.5 Event-to-Event Query

We now describe a variant of the TB query that answers event-to-event queries. Let $q^* = (T_a[i], T_b[j])$ be an event-to-event query. Note that all journeys that end in the target event $T_b[j]$ have the same arrival time. As a result, this algorithm finds the journey (if it exists) that minimizes the number of transfers.

The changes compared to the normal query algorithm (see Algorithm 3) are minimal, namely Q_1 is only filled with $T_a[i, |T_a|]$. As no target labels are required, the first loop in Line 4 over the queue Q_n is also omitted. The minimal arrival time τ_{min}, which is used as target pruning in Line 11, can be replaced by $\tau_{\mathrm{arr}}(T_b[j])$. The rest of the algorithm stays the same.

4 Transfer-Ranked EXploration

4.1 Introduction

The idea behind T-REX (Transfer-Ranked EXploration) is quite simple: when traveling from Paris to Berlin, one typically takes long-distance trains such as the TGV or ICE. If a transfer is required at, e.g., Karlsruhe station, it is made to another long-distance train rather than to a local bus. T-REX formalizes this intuitive insight by assigning an importance to transfers; a more important transfer is necessary to travel far. We call the importance of a transfer $\mathfrak{t} \in \mathfrak{T}$ the *rank* of $\mathfrak{t}$, and is formally defined by the function $r : \mathfrak{T} \to \mathbb{N}_0$. The idea is adapted from ACSA [10], which assigns importance to connections rather than transfers. Connections are the building blocks of ACSA, just as transfers are the building blocks of T-REX. FLASH-TB [5] has demonstrated that storing goal-directed information on transfers, rather than on entire trips or lines, is particularly efficient. Information on transfers is much more fine-grained than information about trips or entire lines, as transfers connect two basic building blocks of timetable; events. The difficulty is to efficiently decide which transfers are important and which ones a query algorithm can skip in order to get provably correct results. Enumerating all optimal journeys between all stops and using these optimal journeys to determine the importance of transfers (the FLASH-TB approach) has quadratic time complexity in the number of stops. The more time is invested during a preprocessing phase in identifying important transfers, the more effectively the search space can be restricted (more on this in Section 5.7). In practice, hand-crafted rules, for instance combining local transport at the source and target to long-distance services, can lead to suboptimal results. Such approaches are particularly prone to failure in cases of major disruptions, such as station closures or track blockages.

Previous partition-based public transit algorithms, such as HypRAPTOR [11] and ACSA (Accelerated Connection Scan Algorithm) [10], explored similar ideas but faced some limitations.

HypRAPTOR [11] is based on the idea of identifying lines that are important for long-distance travel by modeling the timetable as hypergraph, where stops are represented by hyperedges, and lines are vertices. Its preprocessing identifies lines and events that are relevant for cross-cell journeys, and marks them as important. During a query, HypRAPTOR scans only those lines that are either marked as important or are located within the source or target cell. However, even though unmarked events are meant to be skipped, the algorithm still needs to inspect each event to determine whether it is marked. Consequently, much of

the potential speed-up is lost, as valuable time is still spent scanning events that have no effect on the final result. To overcome this, the authors experimented with more sophisticated data structures, such as skip lists [29], to completely bypass irrelevant events. Yet, all proposed variants introduced a higher memory overhead and were less cache-friendly, which limited the speedup.

ACSA [10], which is also based on a multilevel partition of stops, achieves a noticeable speedup, but focused only on the non-Pareto variant of the problem, which is computationally simpler. The algorithm stores for each connection whether it is part of an optimal journey through a cell; and if so, for which cells. A considerable portion of the query time is spent on assembling and sorting the required connections for a given query.

T-REX, on the other hand, benefits from not having to build additional data structures during the query and from the fact that unimportant events make up a very small part of the search space. This is achieved because T-REX only relaxes important transfers, and since important transfers connect two important events, very few unimportant events are scanned.

The first step in the T-REX algorithm is to compute a multilevel nested bipartition of the timetable, and thereafter compute all the transfer ranks in a bottom-up fashion.

We call the computation of ranks *customization*, based on the three-phase model of CCH [9] and CRP [8] for individual transport. Both algorithms are based on three phases: metric-independent precomputation, metric-dependent customization, and the query. The first phase is usually performed infrequently and can therefore take more time (in the order of minutes). Customization, on the other hand, should take only seconds in order to allow e.g., live updates or metric changes to be incorporated quickly. Queries should then be answered in submilliseconds.

4.2 Graph Representation

We represent the timetable as a graph $G_L = (V_L, E_L)$, where vertices correspond to stops, and edges between them represent connections. Stops that are connected by a footpath are contracted into a single vertex. We call this graph *compact layout graph*. Additionally, we weight the vertices and edges using two functions q_L, c_L. Vertices are weighted by the amount of stops they contain, and the edge weight is defined by the number of connections between the stops contained inside the two vertices. After computing a nested bipartition, every stop $p \in \mathcal{S}$ is assigned the cell ID of its representation $v \in V_L$. We note that we model the timetable in the same way as ACSA [10].

As already mentioned, HypRAPTOR [11] and HypTB [20] are also based on partitions, but the modeling of the network as a graph is different than here. The idea of considering lines and stops as hyperedges is algorithmically and intuitively obvious for RAPTOR. However, the approach is not successful with HypTB; it even delivers slower results on some datasets than the baseline TB query. FLASH-TB, on the other hand, models the network in the same way as we do, but without contracting the footpaths (simply called *layout graph*). With this

approach, FLASH-TB is very successful in drastically limiting the search space of the query and achieving sub-milliseconds query times.

4.3 Partitioning

The first step of T-REX is to compute a balanced multilevel nested partition of the compact layout graph G_L. Since the layout graph represents the rough underlying structure of the timetable, a new partition only needs to be calculated when there is a significant timetable change; not a mere delay or cancellation or trips. The goal of this multilevel partition is to compute a balanced cut into two cells over multiple levels. Other works, such as ACSA [10], also evaluate their algorithms with more than two cells per level, but the best query performance has been achieved with a multilevel bipartition. Smaller cuts, i.e., fewer cut edges, allow for faster precomputation (cf. Section 4.5), while the balance ensures that all cells do not become disproportionately large or small. In contrast to road graphs, which appear to have small balanced separators similar to planar graphs (see e.g., [30]), this is not necessarily the case with compact layout graphs.

Metropolitan areas such as Berlin or Paris are densely interconnected internally, while comparatively few long-distance or regional connections exist between metropolitan areas. This structure resembles that of social graphs, which represent social relations between persons or entities: such graphs typically exhibit locally dense structures (clusters) and are sparsely connected across clusters [31]. Since clusters are highly interconnected internally, it is unlikely a small balanced cuts exist within a cluster. Therefore, it is reasonable to avoid cutting through clusters and instead identify separators between clusters.

Social graphs are often *power-law* graphs, meaning that their degree distribution follows a power-law function of the form $f(d) = \alpha d^{\beta}$ where $f(d)$ denotes the number of vertices of degree d and $\beta < 0$. In such graphs, the vast majority of vertices have very low degree, while a small number of vertices have very high degree. These graphs are also referred to as *scale-free* graphs [32].

Although we do not claim that the (compact) layout graph formally satisfies these scale-free graph properties, empirical and structural observations suggest a similar but weaker behavior. In this sense, the compact layout graphs appear to lie between scale-free and planar graphs, such as road networks.

We can therefore exploit these characteristics, for example by using partitioners that have been specifically developed and tuned for social graphs, such as Mt-KaHyPar [33, 34, 35].

The presence of clustered structures in timetable graphs was previously observed by ACSA [10] and Scalable Transfer Patterns [18]; as well as the more theoretical work by Bast [36]. However, ACSA did not exploit any social graph features during partitioning, whereas Scalable Transfer Patterns employed a simple, fast, non-graph-based approach for identifying clusters. More details on this are provided in Section 5.3.

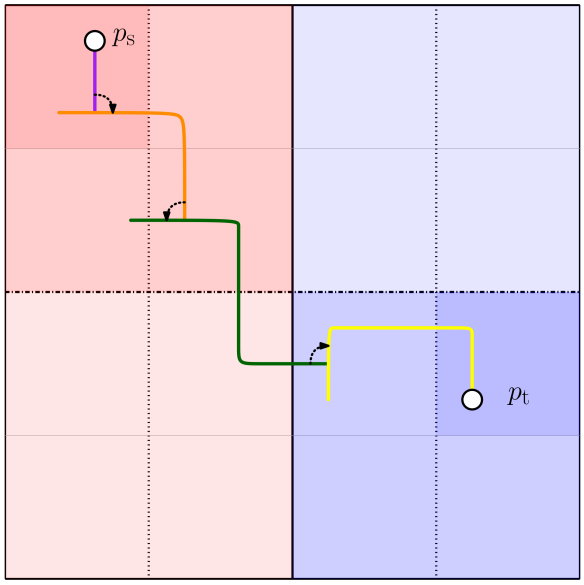


Figure 4.1: The search space of a fictitious query from p_s to p_t is shown for a nested bipartition with four levels. Trips are the colored lines, the transfers are the black dotted arrows. The background is shaded red on the "source side" and blue on the "target side", with the opacity at each point indicating which level of the nested bipartition is being explored. Darker shading means both higher-level and local-level transfers are scanned there; lighter shading means no lower-level transfers are explored.

4.4 Query

Let p_s be the source and p_t the target stop. The T-REX query essentially consists of a TB query, but not all transfers are relaxed. The idea is now to only relax transfers that are "important" enough, depending on p_s and p_t. Whether a transfer is relaxed is computed on the fly during the query using the rank and cell IDs. See Figure 4.1 as an illustration.

Let us assume that the query wants to relax the transfer $\mathfrak{t}$ starting from the stop p. The transfer $\mathfrak{t}$ is only relaxed if $r(\mathfrak{t})$ is greater than or equal to the maximum of both the LCL of p and p_s and the LCL of p and p_t. Hence, $\mathfrak{t}$ is relaxed if the following holds:

$$r(\mathfrak{t}) \geq \max\{\mathrm{LCL}(p, p_s), \mathrm{LCL}(p, p_t)\} \tag{4.4.1}$$

The Equation (4.4.1) leads us to the modified query, as only the `Enqueue` method needs to be adapted; see Algorithm 6, Line 7 for more details. It is important that this additional check is very fast and has a high throughput. See Subsection 4.6.1 for more details.

Algorithm 6: T-REX Query: enqueuing operation.

```
1 Procedure Enqueue(t = (T_a[i], T_b[j]), Q_n)
2     if R(T_b) ≤ j then return                    // trip segment already reached
3     for each T' ⪰ T_b do                          // update reached index
4         if R(T'_b) ≤ j then break
5         R(T'_b) ← j
6     p ← p(T_a[i])
7     if r(t) < max{LCL(p, p_s), LCL(p, p_t)} then return        // LCL check
8     Q_n ← Q_n ∪ {T_b[j, R(T_b) − 1]}
```

4.5 Customization

The customization computes the rank of all transfers and works in a bottom-up fashion; see Algorithm 7. It is based on the event-to-event TB query algorithm.

Given a nested bipartition $\mathcal{H}^2_\ell(G_L)$, a cell $H_k \in \mathcal{H}_l$ on level $l \in [0, \ell)$ and a trip $T \in \mathcal{T}$, we say the event $T[i]$ is an *incoming border event* (IBE) for the cell H_k on level l if the stop of $T[i]$ does not belong to H_k, but the next stop of $T[i+1]$ does. More formally:

$$c_{\text{ID}}\left(p\left(T[i]\right), l\right) \neq c_{\text{ID}}\left(p\left(T[i+1]\right), l\right) = k_{(2)}$$

If $T[i]$ is an IBE for the cell H_k, we call the event $T[i]$ *incoming* for H_k on level l.

Analogously, an event $T[i]$ is called an *outgoing border event* (OBE) for the cell H_k on level l, iff

$$c_{\text{ID}}\left(p\left(T[i]\right), l\right) = k_{(2)} \neq c_{\text{ID}}\left(p\left(T[i+1]\right), l\right).$$

The customization algorithm works as follows (see Algorithm 7):

For every level $l \in [0, \ell)$ (from lowest to highest), all IBEs for cells $H \in \mathcal{H}_l$ are collected (see Line 4), and then a modified event-to-all TB query algorithm is run for each of these IBEs (see Lines 7 to 13). For an IBE $T[i]$, this modified event-to-all TB query algorithm only relaxes transfers that lie within the cell $c_{\text{ID}}\left(p(T[i+1])\right)$ (see Algorithm 8 and Algorithm 9). All the OBEs $T'[k]$ that are scanned for the cell $c_{\text{ID}}\left(p(T[i+1])\right)$ are collected in a set $\mathcal{C} \subseteq \mathcal{E}$. After the search has terminated, the algorithm unpacks every $T[i]$-to-$T'[k]$ journey $J_\mathcal{E}$ found, with $T'[k] \in \mathcal{C}$. Every transfer $\mathfrak{t} \in J_\mathcal{E}$ is *marked* for the current level l, i.e., $r\left(\mathfrak{t}\right) = l + 1$.

During the modified search for an IBE at level l, we only need to consider transfers that both lie within the current cell and have rank l, i.e., transfers that were marked at the previous level (see Algorithm 9, Line 3). This ensures that the algorithm only relaxes relevant transfers, avoiding unnecessary work. Intuitively, a transfer that does not lie on an optimal journey through a child cell cannot appear in any optimal journey through its parent cell.

Another optimization is to avoid collecting all IBEs on every level. Instead, we maintain a list of IBEs and, at each level, remove only those that are no longer IBEs of the current level.

Algorithm 7: T-REX Customization: high-level overview.

```
1  Procedure Customize(H^2_ℓ(S))
2      r(t) = 0 ∀ t ∈ 𝔗
3      for l ∈ [0, ℓ) do
4          Ē ← IBEs on level l
5          for each T[i] ∈ Ē do
6              c ← cell of T[i+1] on level l
7              // init data structures for Scan
8              Q_1 ← {T[i+1, |T|]}
9              C, R ← reset data structures
10             n ← 1
11             while Q_n ≠ ∅ do
12                 Scan(Q_n, C, c, l)
13                 n ← n + 1
14             // assign rank to all transfers of found journeys
15             for each ε ∈ C do
16                 r(t) ← l + 1, ∀ t ∈ T[i]-to-ε subjourney J_ε
```

4.5.1 Partial Proof of Correctness

The correctness of T-REX, as presented in this thesis, is not proven. We prove that T-REX is correct under weaker assumptions by omitting the transfer reduction rules applied during the transfer generation.

Given a query $q = (p_s, p_t, \tau_{\text{dep}})$, we denote by $\mathtt{E}(q)$ the set of scanned events during the TB query algorithm for q. Analogously, for an event-to-event query $q^* = (T_a[i_a], T_b[j_b])$, we define $\mathtt{E}(q^*)$.

We define two orderings of events, one total ordering to describe the order in which events are scanned ($\prec_q$), and one partial ordering to compare two events of the same line at the same stop index ($\preceq_{\mathcal{E}}$).

The first (total) ordering $\prec_q$ is defined as: For two events $\varepsilon_1 \neq \varepsilon_2 \in \mathtt{E}(q)$, we write $\varepsilon_1 \prec_q \varepsilon_2$ iff ε_1 was scanned before ε_2 during the processing of q.

For the second (partial) ordering $\preceq_{\varepsilon}$, consider two events $\varepsilon_1 = T_1[i]$ and $\varepsilon_2 = T_2[j]$ with $i, j \in \mathbb{N}$. If $i = j$ and T_1 and T_2 belong to the same line, then we write $\varepsilon_1 \preceq_{\mathcal{E}} \varepsilon_2$ iff $T_1 \preceq T_2$. Otherwise, ε_1 and ε_2 are not comparable.

Proposition 4.5.1 (Queue-Prefix-Optimality)**.** *Let $J = \langle T_1[i_1, j_1], \mathfrak{t}_1, \ldots, \mathfrak{t}_{n-1}, T_n[i_n, j_n] \rangle$ be an optimal journey computed by the TB query algorithm for the query $q = (p_s, p_t, \tau_{\text{dep}})$. For $k \in [1, n-1]$, consider the transfer*

$$\mathfrak{t}_k = (T_k[j_k], T_{k+1}[i_{k+1}]) \in J.$$

Algorithm 8: T-REX Customization: trip scanning operation for the IBE $T[i]$ of cell ID c at level l.

```
1  Procedure Scan(Q_n, C, c, l)
2      Q_{n+1} ← ∅
3      for each T[j,k] ∈ Q_n do
4          for i from j to k do
5              // mark T[i] if outside
6              if !InSameCell(p(T[i]), c, l) then C ← C ∪ {T[i]}
7      for each T[j,k] ∈ Q_n do
8          for i from j to k do
9              for each t = (T[i], T'[i']) ∈ 𝔗 do          // relax transfers
10                 Enqueue(t, c, l, Q_{n+1})
```

Algorithm 9: T-REX Customization: enqueuing operation.

```
1  Procedure Enqueue(t = (T_a[i], T_b[j]), c, l, Q_n)
2      if R(T_b) ≤ j then return                  // trip segment already reached
3      if r(t) < l then return                    // unimportant transfer
4      if !InSameCell(p(T_b[j]), c, l) then return   // not in this cell
5      Q_n ← Q_n ∪ {T_b[j, R(T_b) − 1]}
6      for each T'_b ⪰ T_b do                     // update reached index
7          if R(T'_b) ≤ j then break
8          R(T'_b) ← j
```

Then, for every event $\varepsilon \in \mathtt{E}(q)$ which is scanned in the k-th round with $\varepsilon \prec_q T_k[j_k]$, there does not exist a q-feasible journey

$$J' = \left\langle T'_1[i'_1, j'_1], \mathfrak{t}'_1, \ldots, \mathfrak{t}'_{n-1}, T_n[i'_n, j_n] \right\rangle$$

with $\mathfrak{t}'_k = \left(\varepsilon, T'_{k+1}[i'_{k+1}]\right)$ as its k-th transfer.

Proof by Contradiction. Assume such a q-feasible journey J' exists. Among all such journeys, choose J' such that for every $l > k$, all events of the trip segment $T'_l[i'_l, j'_l]$ are the earliest reachable events of line $L(T'_l)$ during round l.

Let $m \in \mathbb{N}$ be the first index such that both journeys share the same suffix starting at m, i.e., for all $l \geq m$, it holds that $T_l[i_l, j_l] = T'_l[i'_l, j'_l]$. Then, since $\mathfrak{t}'_k$ is relaxed before $\mathfrak{t}_k$, every subsequent event in a trip segment $T'_l[i'_l, j'_l] \in J'$ with $l \in (k, m)$ will be scanned before the corresponding trip segment $T_l[i_l, j_l] \in J$, because every queue is filled in FIFO order. Because the events of trip segment $T'_l[i'_l, j'_l]$ are the earliest reachable events of line $L(T'_l)$ in round l, they are scanned. As the event $T_n[j_n]$ is reached via the transfer $\mathfrak{t}'_{n-1}$, which is relaxed before $\mathfrak{t}_{n-1}$, the parent pointer of the target label for p_t with n trips is not set

to the trip segment $T_n[i_n, j_n] \in J$. Hence, J is not unpacked by the algorithm. This is a contradiction to our assumption that the TB query algorithm returned J. □

Let $\mathfrak{T}^*$ be the *full* transfer set generated by TB before applying the U-turn and latest-exit reduction rules. We now prove the correctness of T-REX when using $\mathfrak{T}^*$.

In the following, the terms trip segment and queue element are used interchangeably. When we say that a TB-based query algorithm "scans" a trip segment $T[i, j]$, this means that a trip segment $T[i', j']$ is in the queue, such that $1 \leq i' \leq i < j \leq j' \leq |T|$.

Lemma 4.5.2. *Let $J = \langle T_1[i_1, j_1], \mathfrak{t}_1, \ldots, \mathfrak{t}_{n-1}, T_n[i_n, j_n] \rangle$ denote an optimal journey computed by the TB query algorithm for the query $q = (p_s, p_t, \tau_{\text{dep}})$ using the full transfer set $\mathfrak{T}^*$.*

Consider a $T_a[i_a]$-to-$T_b[j_b]$ subjourney $J_\mathcal{E}$ of J. Given $q^* = (T_a[i_a], T_b[j_b])$ and the full transfer set $\mathfrak{T}^*$, the event-to-event TB query algorithm scans every trip segment $T[i, j] \in J_\mathcal{E}$ and, for all $k \in [1, |J_\mathcal{E}|)$, the algorithm calls Enqueue *for the transfer $\mathfrak{t}_k$.*

Proof by Contradiction. Consider the first trip segment $T_k[i_k, j_k] \in J_\mathcal{E}$ which is not scanned by the event-to-event query. Because the event-to-event TB query algorithm enqueues the trip segment $T_a[i_a, |T_a|]$; hence $k > 1$.

Observation (1): If a trip segment $T[i, j]$ is scanned by a TB based query algorithm, Enqueue is called for all the outgoing transfers of that trip segment, i.e., transfers departing from an event $T[l]$, for all $l \in [i, j]$.

Additionally, because all previous trip segment $T_l[i_l, j_l] \in J_\mathcal{E}$, with $l \in [1, k)$, are scanned by the event-to-event TB query algorithm, it follows with Observation (1) that Enqueue has been called for all $\mathfrak{t}_l = (T_l[j_l], T_{l+1}[i_{l+1}])$.

See Figure 4.2. The trip segment $T_k[i_k, j_k] \in J_\mathcal{E}$ is not being scanned by the event-to-event TB query algorithm because either

(i) an earlier event $T'_k[i_k]$, with $T'_k \prec T_k$, has been reached before $T_k[i_k]$, or

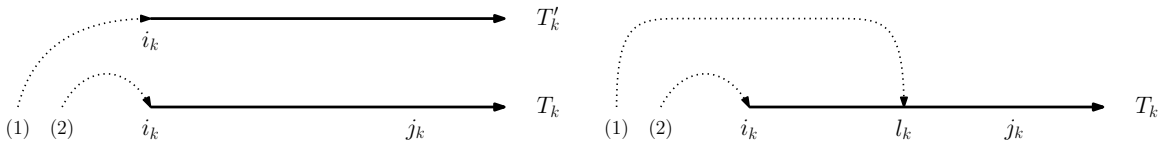
(ii) an event $T_k[l_k]$, with $l_k \in (i_k, j_k]$, has been reached before $T_k[i_k]$. Note: Although all events $\varepsilon \in T_k[i_k, j_k]$ are scanned, the trip segment itself is not scanned within the same queue element, because it is "split" at position l_k.

Let $T^*_k[i^*_k] \in \mathcal{E}$ be the reason why $T_k[i_k, j_k]$ is not scanned during the event-to-event TB query algorithm, and $J^*_\mathcal{E}$ the $T_a[i_a]$-to-$T^*_k[i^*_k]$ subjourney found by the event-to-event TB query algorithm.

All trip segments or events marked by $*$ refer to elements of $J^*_\mathcal{E}$.

Observation (2): The TB query algorithm did not reach $T^*_k[i^*_k]$ before $T_k[i_k]$, as otherwise the trip segment $T_k[i_k, j_k]$ would not be part of the returned journey J.

The following leads to a contradiction to Observation (2), which means no such event $T^*_k[i^*_k]$ is reached before $T_k[i_k]$ during the event-to-event query.



(a) Case (1) (b) Case (2)

Figure 4.2: The figure shows trip segments and transfers, with trip segments represented by a thicker, solid line and transfers represented by a dotted line. The numbering on the transfers indicates the order in which these transfers are relaxed. The left hand side denotes that an earlier event $T'_k[i_k]$, with $T'_k \prec T_k$, has been reached before $T_k[i_k]$. The right hand side shows the case in which an event $T_k[l_k]$, with $l_k \in (i_k, j_k]$, was reached before $T_k[i_k]$.

We denote by T_r the last trip, which both $J_\mathcal{E}$ and $J^*_\mathcal{E}$ share, with $T_r[i_r, j^*_r] \in J^*_\mathcal{E}$ and $j^*_r \in (i_r, j_r)$. During the TB query algorithm, Enqueue is called for the transfer $\mathfrak{t}^*_r = \left(T_r[j^*_r], T^*_{r+1}[i^*_{r+1}]\right) \in J^*_\mathcal{E}$. It could be however, that the reached index prunes the event $T^*_{r+1}[i^*_{r+1}]$, because an even earlier event $T^\dagger_{r+1}[i^*_{r+1}] \prec_\mathcal{E} T^*_{r+1}[i^*_{r+1}]$ had already been reached. Nevertheless, a trip segment starting from an event $T^\dagger_{r+1}[i^*_{r+1}] \preceq_\mathcal{E} T^*_{r+1}[i^*_{r+1}]$ is added to the next queue.

Starting from $T^\dagger_{r+1}[i^*_{r+1}]$, we prove by induction that the TB query algorithm reaches trip segments $T^\dagger_o[i^*_o, j^*_o] \preceq T^*_o[i^*_o, j^*_o] \in J^*_\mathcal{E}$, with $o \in [r+1, k]$, meaning it reaches either the same trip segments, or earlier ones.

Base case $o = r + 1$: Because the TB query algorithm reached $T^\dagger_{r+1}[i^*_{r+1}]$, it reached all subsequent events along the trip (possibly even earlier ones).

Inductive step $o - 1 \to o$: From $T^\dagger_{o-1}[i^*_{o-1}, j^*_{o-1}]$, transfers $\mathfrak{t}^*_{o-1} \in \mathfrak{T}^*$ exist towards all earliest reachable events of line $L\left(T^\dagger_o\right)$. The TB query algorithm therefore calls Enqueue for each such transfer. One of those transfers leads to $T^\dagger_o[i^*_o]$, and hence all events along $T^\dagger_o[i^*_o, j^*_o]$ (or possibly earlier ones) are reached.

However, this leads to a contradiction to Observation (2), as now an event $T^\dagger_k[i^*_k] \preceq T^*_k[i^*_k]$ was reached before $T_k[i_k]$, because $T^\dagger_{r+1}[i^*_{r+1}] \prec_q T_{r+1}[i_{r+1}]$. □

If these properties hold, then the following theorems follow:

Theorem 4.5.3 (Event-To-Event Subjourney Closure)**.** *Let J be an optimal journey computed by the TB query algorithm for a query $q = (p_s, p_t, \tau_{\text{dep}})$ and $J_\mathcal{E}$ the $T_a[i_a]$-to-$T_b[j_b]$ subjourney $J[T_a[i_a], T_b[j_b]]$ of J.*

The event-to-event TB query algorithm for $q^* = (T_a[i_a], T_b[j_b])$ finds $J_\mathcal{E}$ as an optimal journey.

Proof. We write

$$J_\mathcal{E} = \langle T_1[i_1, j_1], \mathfrak{t}_1, \ldots, \mathfrak{t}_{l-1}, T_l[i_l, j_l] \rangle ,$$

with $T_1[i_1] = T_a[i_a]$ and $T_l[j_l] = T_b[j_b]$.

By Lemma 4.5.2, the event-to-event TB query algorithm scans every trip segment and calls Enqueue for every transfer of $J_{\mathcal{E}}$.

We show by induction over the number of trips $k \in \mathbb{N}^+$ in $J_{\mathcal{E}}$, that for every scanned event $\varepsilon \in T^*[i^*, j^*] \in Q_k$ during the event-to-event TB query algorithm, such that $\varepsilon \prec_{q^*} T_k[j_k] \in J_{\mathcal{E}}$, there does not exist a q^*-feasible journey $J'_{\mathcal{E}}$ which uses the transfer $\mathfrak{t}'_k = (\varepsilon, \cdot)$ as its k-th transfer. Thus, the journey that is found (and extracted) by the event-to-event TB query algorithm is $J_{\mathcal{E}}$.

Base case: $k = 1$*:* The first queue of the event-to-event TB query algorithm is initialized with $Q_1 = \{T_a[i_a, |T_a|]\}$. Every event $\varepsilon \in T_a[i_a, j_a - 1]$ is scanned by both algorithms before $T_a[j_a]$, i.e., $\varepsilon \prec_q T_a[j_a] \wedge \varepsilon \prec_{q^*} T_a[j_a]$. Hence, by Proposition 4.5.1, there does not exist such a q^*-feasible $T_a[i_a]$-to-$T_b[j_b]$ journey $J'_{\mathcal{E}} = J[T_a[i_a], T_b[j_b]]$ with the first transfer departing from an event $\varepsilon \in T_a[i_a, j_a - 1]$.

Induction step: $k - 1 \to k$*:* Consider any trip segment $T_k^\dagger[i_k^\dagger, j_k^\dagger] \in Q_k$, such that $T_k^\dagger[j_k^\dagger] \prec_{q^*} T_k[i_k]$. Note that $T_k^\dagger$ could be the same trip as T_k, but this implies $j_k < i_k^\dagger$. The trip segment $T_k^\dagger[i_k^\dagger, j_k^\dagger]$ was enqueued because a transfer $\mathfrak{t}^\dagger = \left(T_{k-1}^\dagger[j_{k-1}^\dagger], T_k^\dagger[i_k^\dagger]\right)$ was relaxed in the previous round. Because we assumed $T_k^\dagger[i_k^\dagger] \prec_{q^*} T_k[i_k]$ and the queue is filled in FIFO order, it follows that

$$T_{k-1}^\dagger[i_{k-1}^\dagger] \prec_{q^*} T_{k-1}^\dagger[j_{k-1}^\dagger] \prec_{q^*} T_{k-1}[i_{k-1}].$$

By induction, we know that no q^*-feasible event-to-event journey $J[T_a[i_a], T_b[j_b]]$ exists that uses the event $T_{k-1}^\dagger[j_{k-1}^\dagger]$ as the departing event for the $(k-1)$-th transfer. Hence, no event $\varepsilon \in T_k^\dagger[i_k^\dagger, j_k^\dagger]$ can be a part of any q^*-feasible event-to-event journey $J[T_a[i_a], T_b[j_b]]$ journey; especially not as departing event of the k-th transfer.

□

Theorem 4.5.4 (T-REX Customization Correctness)**.** *Given a nested bipartition $\mathcal{H}_\ell^2(G_L)$. Let J be an optimal journey computed by the TB query algorithm for a query $q = (p_s, p_t, \tau_{\text{dep}})$. After the T-REX customization, every transfer $\mathfrak{t}$ in J fulfills Equation (4.4.1).*

Proof. Consider any transfer $\mathfrak{t} = (T_m[j_m], T_n[i_n])$ in J. Because footpaths are contracted during the construction of the layout graph G_L, it follows that $c_{\text{ID}}(p(T_m[j_m])) = c_{\text{ID}}(p(T_n[i_n]))$. Let $r = \max\{\text{LCL}(p(T_m[j_m]), p_s), \text{LCL}(p(T_m[j_m]), p_t)\}$ and $J_{\mathcal{E}} = J[T_a[i_a], T_b[j_b]]$ be the maximal event-to-event subjourney of J such that $J_{\mathcal{E}}$ uses $\mathfrak{t}$ and $J_{\mathcal{E}}$ is entirely inside the cell $c_{\text{ID}}(p(T_m[j_m]), r)$.

During the customization of cell $c_{\text{ID}}(p(T_m[j_m]), r)$, an event-to-event subjourney $J_{\mathcal{E}}^*$ from $T_a[i_a]$ to $T_b[j_b]$ is found by the T-REX customization and for every transfer $\mathfrak{t}^* \in J_{\mathcal{E}}^*$, the rank is updated to $r(\mathfrak{t}^*) = r + 1$. Without the pruning rule in the modified Enqueue method (see Algorithm 9, Line 3), the customization behaves like an event-to-event TB query algorithm limited to a cell. Thus, by Theorem 4.5.3, $J_{\mathcal{E}} = J_{\mathcal{E}}^*$, and hence Equation (4.4.1) holds for $\mathfrak{t}$.

We only need to show that Line 3 does not prune any transfer of $J^*_{\mathcal{E}}$, and prove it by induction over the levels $l \geq 0$.

Base case $l = 0$*:* For level $l = 0$, the pruning rule does not apply. Hence the customization on the lowest level is correct.

Inductive step $l \to l + 1$: We can subdivide $J^*_{\mathcal{E}}$ into many smaller event-to-event subjourneys that lie completely in one of the subcells of the current cell. Each one of those event-to-event subjourneys can again be represented as event-to-event subjourney $J[T'_a[i'_a], T'_b[j'_b]]$, and by induction, the customization (at the previous level l) finds and marks every transfer in this journey, i.e.,

$$\forall\, \mathfrak{t} \in J[T'_a[i'_a], T'_b[j'_b]] : r(\mathfrak{t}) = (l-1) + 1 = l.$$

Hence, the additional pruning rule in Line 3 holds for any level $l \geq 0$. □

So far, the proof has been shown under weaker assumptions, which excluded the U-turn and latest-exit pruning rules for the transfer set. The next step is to show that both reduction rules preserve the properties of Lemma 4.5.2. Once these properties are proven, the correctness of T-REX follows.

However, proving that these reduction rules individually preserve these properties, is not straightforward, as there does not exist a formal correctness proof for TB, and, more importantly, the order in which the reduction rules are applied is crucial.

Ordering of Reduction Rules. It is important that the U-turn rule is applied before the latest-exit rule. Consider the example in Figure 4.3.

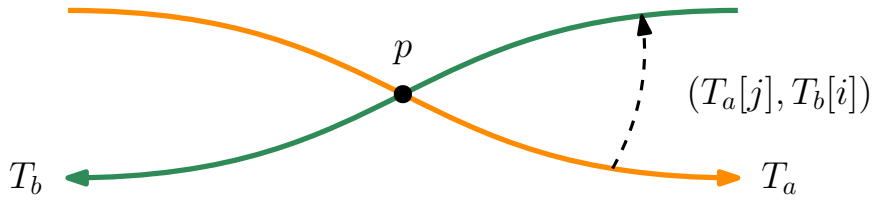


Figure 4.3: This figure shows the structure of the U-turn rule, with $p = p(T_a[j-1]) = p(T_b[i+1])$ and $\tau_{\text{arr}}(T_a[j-1]) \leq \tau_{\text{dep}}(T_b[i+1])$.

If the U-turn rule is applied first, the transfer $(T_a[j], T_b[i])$ is removed because the time constraint at stop p is satisfied. However, if the latest-exit rule is applied first, the transfer $(T_a[j-1], T_b[i+1])$ at stop p would be removed instead. Since the U-turn rule only verifies whether a transfer would be *feasible* in terms of transfer time, but does not check whether the corresponding transfer actually exists, applying it after the latest-exit rule would incorrectly delete the latter transfer; hence both transfers would be removed. To solve this problem, Line 6 in Algorithm 2 must therefore be updated as follows:

$$\text{valid} \leftarrow (T_a[j-1], T_b[i+1]) \in \mathfrak{T}$$

This condition is stronger than the previous one, since the existence of a transfer in $\mathfrak{T}$ already guarantees that the corresponding time constraint is satisfied.

Taken together, these observations show that proving the preservation properties of the U-turn and latest-exit rules is non-trivial. Consequently, formal proofs of these conjectures remain as open questions.

Conjecture 4.5.5. *Given the full transfer set $\mathfrak{T}^*$, reducing it using the U-turn rule preserves the properties of Lemma 4.5.2.*

Conjecture 4.5.6. *Given the full transfer set $\mathfrak{T}^*$, reducing it using the latest-exit rule preserves the properties of Lemma 4.5.2.*

4.5.2 Parallelization

Since the customization runs an event-to-event query for every IBE of every cell on every level, it is not clear as how to parallelize the total work optimally. On every level $l \in [0, \ell)$, there are multiple cells $H_0, H_1, \cdots \in \mathcal{H}_l$, and within each cell H_i are multiple IBEs that need to be processed. Two parallelize schemes seem obvious for a given level l: Either the cells $H_0, H_1, \cdots \in \mathcal{H}_l$ are distributed to all the threads, and hence one worker is processing all IBEs of its cell, or all threads process one cell at a time. The latter results in all threads processing the IBEs of one cell at a time. Both have disadvantages.

If each thread works on a separate cell, it can lead to many threads waiting for the work of the others, as they have to work on "difficult" cells. On the other hand, if all threads cooperate on one cell at a time, the relative overhead of distributing and synchronizing the work becomes significant, since processing a single IBE is comparatively lightweight.

We consider a slightly different parallelization scheme: All IBEs per level are collected together with the corresponding cell, and this is distributed among all the available threads. This can result in one thread working on multiple cells. This approach avoids the synchronization bottleneck of the previous scheme, as threads do not need to wait for an entire cell to finish before continuing with the next work.

The precomputation scheme of ACSA also follows a bottom-up strategy. The authors propose a more advanced parallel scheme that models the dependencies between all cells as a directed tree, where each edge represents a parent-child relationship between cells. Every vertex keeps track of the number of unfinished children; an *internal counter*. In this tree, a cell becomes active once all of its children have been processed, i.e., when its internal counter drops to zero. The parallelization proceeds in a manner similar to topological sorting: t threads are launched, each selecting an active vertex to process or assisting another thread with its current work. As soon as a thread finishes processing a cell, the internal

counter of its parent is decreased accordingly. When a parent's internal counter reaches zero, it becomes active and eligible for processing.

We chose a simpler parallelization approach, primarily due to its straightforward implementation and also because it already provides strong speedup and efficiency (cf. Section 5). Developing a more sophisticated parallel customization remains future work.

4.6 Advanced Techniques and Extensions

In this section, we describe how to efficiently compute the LCL using bit manipulation, and we also discuss two extensions that could be developed in future work.

4.6.1 Bit Manipulation

The representation of the cell ID as an integer has many advantages, because on the one hand the representation is very compact and memory efficient. This means that many cell IDs fit into one cache line, which enables efficient scanning via consecutive memory. On the other hand, the manipulation and calculation of hierarchy operations, such as the calculation of LCL, is very efficient and has a high throughput, as this can be calculated very quickly through bit manipulation.

For example, computing the LCL of two vertices $u, v \in V$ with the corresponding cell IDs $c_{\text{ID}}(u), c_{\text{ID}}(v) \in \left[0, 2^{\ell}\right)$, $\ell \in \mathbb{N}^{+}$, can be done the following way:

$$\text{LCL}(u, v) = \ell - \text{std::countl\_zero}\left(c_{\text{ID}}(u) \oplus c_{\text{ID}}(v)\right)$$

First, we XOR both cell IDs, which results in an integer with a 1 on each level where both vertices are in different cells. In order to then get the highest level in which they differ, we can use the standardized `C++20` function std::countl_zero from the `<bit>` header, which counts the number of leading zeros in an unsigned integer.[1] Subtracting the result from the total number of levels ℓ yields the LCL value.

As an illustration, assume the nested bipartition shown in Figure 3.1 and suppose $c_{\text{ID}}(u) = 001_{(2)}$ and $c_{\text{ID}}(v) = 101_{(2)}$. Taking the bitwise XOR gives $c_{\text{ID}}(u) \oplus c_{\text{ID}}(v) = 100_{(2)}$, which has no leading zeros in its 3-bit representation (3 bits due to 3 levels). Hence, the $\text{LCL}(u, v)$ is $3 - 0 = 3$ (as u and v do not share any cells).

In the following, the operators $\&\&$, $\oplus$, and $\gg$ denote the *programming* logical AND, bitwise XOR, and right-shift operators, respectively, whereas $\wedge$ denotes the *mathematical* logical AND operator.

[1] On compilers that do not yet support C++20 or `<bit>`, this can also be implemented using compiler intrinsics such as `__builtin_clz` (GCC/Clang) or `_BitScanReverse` (MSVC).

We can efficiently implement Line 7 in Algorithm 6 using the following approach. Let $\mathcal{H}^2(G_L)$ and $p, p_s, p_t \in \mathcal{S}$ be given, as well as a transfer $\mathfrak{t}$ starting at p:

$$[(c_{\text{ID}}(p) \oplus c_{\text{ID}}(p_s)) \gg r(\mathfrak{t})] \neq 0 \;\;\&\&\;\; [(c_{\text{ID}}(p) \oplus c_{\text{ID}}(p_t)) \gg r(\mathfrak{t})] \neq 0 \tag{4.6.1}$$

Instead of calculating the two LCL values, we implement the comparisons implicitly, using a right shift. We shift the XOR result exactly as many bits to the right as the rank of the transfer. Note that this condition is the negated variant of Equation (4.4.1).

Consider the condition (4.6.1) evaluating to false, i.e., the transfer $\mathfrak{t}$ should not be pruned and fulfills Equation (4.4.1). Since the condition is symmetric, we show the equivalence only for the left-hand side; the same reasoning applies to the right-hand side.

$$\begin{aligned} & [(c_{\text{ID}}(p) \oplus c_{\text{ID}}(p_s)) \gg r(\mathfrak{t})] = 0 \text{ (False)} \\ \iff & (\ell - \text{std::countl\_zero}(c_{\text{ID}}(p) \oplus c_{\text{ID}}(p_s))) \leq r(\mathfrak{t}) \\ \iff & \text{LCL}(p, p_s) \leq r(\mathfrak{t}) \end{aligned}$$

and thus, Equation (4.4.1) holds, since

$$\begin{aligned} & r(\mathfrak{t}) \geq \text{LCL}(p, p_s) \wedge r(\mathfrak{t}) \geq \text{LCL}(p, p_t) \\ \iff & r(\mathfrak{t}) \geq \max\{\text{LCL}(p, p_s), \text{LCL}(p, p_t)\}. \end{aligned}$$

4.6.2 Transfer Shortcuts

CRP [8] (Customizable Route Planning) is a speedup technique for shortest-path queries on road networks that relies on a multilevel partitioning of the graph. To accelerate Dijkstras algorithm [14], CRP avoids exploring unimportant cells by introducing shortcuts between all boundary vertices $u, v \in V$. These shortcuts preserve the shortest-path distances of the original graph between boundary vertices, allowing the algorithm to skip the cell in a single step.

During preprocessing, for each cell, the shortest paths between all pairs of boundary vertices are computed. However, the paths themselves are not stored; only their lengths $\text{dist}(u, v)$ are kept and set as edge weights of the corresponding shortcut from u to v. As a result, queries can simply relax these shortcut edges directly, without having to traverse the underlying path inside the cell.

Similar to CRP, it would be possible for the T-REX algorithm to create *transfer shortcuts* that span an entire cell. For every cell, a transfer shortcut connects an incoming border event ε_A to an outgoing border event ε_B, and stores the minimal number of transfers k needed to get from ε_A to ε_B. However, in contrast to shortest-path queries on road networks, our problem requires computing a Pareto set, which involves determining multiple optimal journeys; instead of a single one. T-REX, TB [1], and RAPTOR [3] break down this additional

complexity by working their way through the network in rounds. For each round $i \in \mathbb{N}$, an optimal journey J is found that minimizes the arrival time with i trips (if such a journey exists). Transfers explored in round i enqueue trip segments that are scanned in the next round $i+1$. A transfer shortcut t from ε_A to ε_B represents the minimum number of transfers k required to reach ε_B starting from ε_A. Consequently, if such a shortcut t is relaxed during round i, the associated trip segment at ε_B must be enqueued for round $i+k$.

The TB query algorithm does not maintain multiple queues Q_i for each round $i \in \mathbb{N}$, but instead uses a single preallocated queue Q^+. This improves performance, primarily due to better cache locality and the avoidance of dynamic memory reallocations during the query. Since each trip segment is enqueued at most once, the number of events $|\mathcal{E}|$ is an upper bound for Q^+.

To perform the required queue operations efficiently, two pointers are used to indicate the current positions for push and pop. As a consequence, new elements can only be inserted into the queue for processing in the next round. In other words, inserting items for later rounds would require multiple queues, which would either increase memory consumption or reduce performance due to additional memory allocations during the query.

However, an efficient query algorithm capable of handling such shortcuts could be promising for future work.

4.6.3 Delays and Cancellations

It is important for modern journey planning algorithms to take real-time data into account. RAPTOR and CSA work directly on the timetable, and therefore do not require an extra update phase to be able to incorporate real-time data. TB may have to recalculate transfers because some transfers are no longer valid in terms of transfer time, or because better, i.e., earlier trips are now reachable for some transfers. Processing a single real-time message can therefore take some time, which is why such real-time information is processed in batches. See [37] for more details about these update phases of TB.

We distinguish between real-time information, such as delays and cancellations, and fundamental modifications to the timetable. Since a computed partition of the stops may degenerate when lines or stops change, only minor timetable updates can be incorporated without having to recompute the partition. As shown in Section 5.3, computing a nested bipartition of all stops requires only a few seconds. Therefore, even substantial timetable changes can be integrated quickly.

Of course, T-REX can re-run the entire customization process whenever the timetable changes, but doing so would be wasteful for large networks. Furthermore, the insight that e.g., a delay of a train does not require the customization of all cells and all incoming events, but only a part, is important to avoid a complete customization. In practice, real-time data is typically received at regular intervals (e.g., every 30 seconds). Such data contains information about currently operating vehicles and, in some cases, about trips that are

about to depart. This means, in particular, that such real-time information usually only provides information about a small time interval. Although this information usually refers to a relatively short time window, its effects may extend well beyond that. For example, a long-distance train that travels across Europe all day can encounter such delays in the morning that precomputed transfers to other vehicles along the entire journey (and thus the entire day) are no longer valid.

We now describe how one can adapt the customization for small changes so as to avoid having to completely compute everything from the ground up. Instead, we customize only the relevant cells in a bottom-up fashion, stopping once a cell no longer experiences a rank change in its transfers. This is because as soon as there are no changes, all cells above it (and its transfers) would not experience any changes either.

Let us assume a local bus is delayed and all new transfers have been computed. Several outcomes are possible: The delay of the bus could have no effect on surrounding transfers because the delay is not significant enough. However, it could also be that the delay changes the (true) ranks of several transfers, e.g., because an optimal journey passing through a cell has changed and therefore other transfers are now considered "important". In the worst-case scenario, this change could propagate all the way to the top, to the highest cell, and all cells on the path to the top would have to be customized.

A trip can pass through several cells, which may involve the recalculation of all the cells crossed. However, only the cells in which the transfers to and from the trip have changed need to be recustomized. This is due to the fact that transfers never span across multiple cells, as footpaths are contracted during the creation of G_L. For example, let T be a trip with the following stop sequence:

$$\Pi(T) = \langle p_A, p_B, p_C, p_D \rangle ,$$

passing through two cells such that p_A, p_B are in the first cell and p_C, p_D are in the second cell. If T experiences a delay at p_C, this delay does not affect the first cell; only the second cell needs to be re-customized.

We did not evaluate this scenario experimentally; the discussion above is purely theoretical. However, as the experiments on customization in Section 5 demonstrate, the entire process takes only a couple of minutes. Furthermore, when the customization is restricted to a two-hour time window (the one with the highest number of active trips) on the German network, the total runtime is reduced to less than two seconds. Therefore, it is reasonable to expect that an incremental, smaller-scale customization would be significantly faster, even with the additional transfer update step.

5 Experiments

We evaluate all algorithms on real-world datasets, which are freely available in the form of the General Transit Feed Specification (GTFS) data. All algorithms are implemented in `C++` and compiled with a `g++` compiler with optimizations turned on (`-march=native -O3`). The code for all algorithms (except TB-CST, CSA and ACSA) is publicly available[1][2]. The programs were executed on two machines, which were also used for the FLASH-TB experiments [5]. All precomputations were run on a machine with two 64-core AMD Epyc Rome-7742 CPUs clocked at $2.25\,\mathrm{GHz}$, with a boost frequency of $3.4\,\mathrm{GHz}$, $1\,024\,\mathrm{GB}$ of DDR4-3200 RAM, and $256\,\mathrm{MB}$ of L3 cache. All queries were performed on a machine with two 8-core Intel Xeon Skylake SP Gold-6144 CPUs clocked at $3.5\,\mathrm{GHz}$, with a boost frequency of $4.2\,\mathrm{GHz}$, $192\,\mathrm{GB}$ of DDR4-2666 RAM, and $24.75\,\mathrm{MB}$ of L3 cache. We refer to the machine used for precomputation as `Epyc`, the other as `Xeon`.

5.1 Datasets

We evaluate all algorithms on several different datasets, based on real GTFS data. GTFS[3] is an open standard designed for sharing essential transit system information with riders. The datasets include metropolitan regions (Paris and Berlin), country-wide networks (Switzerland, Great Britain, and Germany), as well as a Europe-wide network; cf. Table 5.1.

The latter was "artificially" created by merging GTFS datasets from all European countries, using the tool gtfstidy[4]. Due to the sheer size, only June 2024 was extracted from all GTFS datasets. Stops closer than $2.5\,\mathrm{m}$ were merged, and footpaths were generated between all pairs of stops closer than $100\,\mathrm{m}$. A walking speed of $1.43\,\mathrm{m/s}$ was used, corresponding to the average human walking speed. The Europe-wide dataset includes $98\,374$ bus lines, $32\,178$ train lines, $9\,320$ subway lines, 920 ferry lines, 622 monorail lines, 19 trolleybus lines and 7 aerial lift lines. For this dataset, additional preprocessing steps were necessary, as merging GTFS feeds from multiple sources is non-trivial. Inconsistencies such as duplicated or overlapping stops from different agencies had to be resolved. The other data sets are significantly cleaner and more consistent, as they are provided by a company or agency.

[1] `https://github.com/PatrickSteil/TREX`
[2] `https://github.com/TransitRouting/FLASH-TB`
[3] `https://gtfs.org/`
[4] `https://github.com/patrickbr/gtfstidy`

The datasets for the European countries were extracted from Transitous[5], and the Germany dataset was extracted from gtfs.de[6]. For the remaining datasets (Paris, Berlin, Switzerland, and Great Britain), the data were similarly extracted from the respective official GTFS feeds. Footpaths for these datasets were taken directly from the GTFS data, whereas for the Europe-wide dataset, additional footpaths were generated during preprocessing.

The stops of each datasets are shown in Figure 5.1 and 5.2. The locations of stops are shown in geographical coordinates (WGS 84 / EPSG:4326[7]). Each sub-figure shows a the stops of the network filtered to a dataset-specific bounding box, chosen to include the core service area (and nearby suburbs) while excluding distant outliers. This filtering is applied only for visualization purposes in the figures.

An interesting aspect of the Europe dataset is the varying level of detail across different countries, for example in terms of long-distance trains versus local traffic. This variation arises because the dataset was derived from multiple sources, each with a different level of granularity.

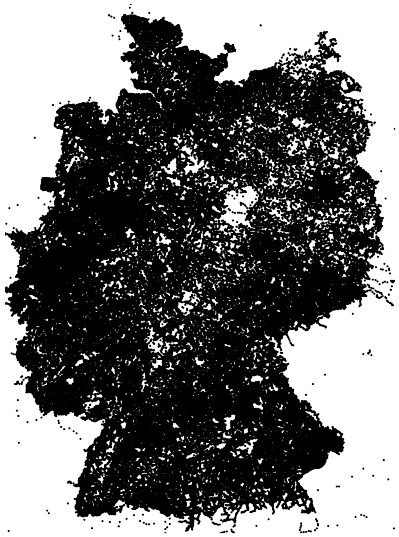
Note that this Europe dataset is the largest dataset used in research, as far as we know. Bauer et al. performed experiments on an European timetable in [15]. However, their extracted "condensed graph" (in size similar to the *layout graph*) was much smaller, with around $30\,000$ vertices and around $87\,000$ edges, whereas our compact layout graph for the Europe instance has $778\,407$ vertices and $2\,298\,358$ edges. Another study that evaluates the algorithm on a European network is [24]. However, their dataset contains only long-distance trains and has approximately $31\,000$ stops and $170\,000$ trips, which is comparable to our Berlin instance.

[5] `https://transitous.org/sources/`
[6] `https://gtfs.de/`
[7] `https://geopandas.org/en/stable/docs/user_guide/projections.html`

Table 5.1: An overview of the networks on which we performed our experiments. Preprocessing for the TB transfer generation is shown as $[\mathrm{mm{:}ss}]$ and was performed with 128 threads. The number of TB transfers, the size of the compact layout graph G_L and the size of the time-expanded G^{ε} and time-dependent G^{δ} graph are also shown. Note that GB denotes Great Britain.

	Europe	Germany	GB	Switzerland	Paris	Berlin
Stops	1 346 013	435 550	260 150	29 045	41 757	25 843
Stop events	107 252 199	30 680 868	19 692 037	5 032 795	4 636 238	3 318 251
Lines	384 075	202 238	36 071	15 967	9 558	10 891
Trips	4 848 876	1 547 126	506 704	319 159	215 526	146 897
Footpaths	1 752 418	1 116 976	345 594	22 186	445 912	29 428
TB $\lvert\mathfrak{T}\rvert$	269 766 664	59 181 359	31 443 687	8 075 627	23 284 123	6 104 156
TB prepro.	08:55	01:23	00:24	00:04	00:17	00:07
$\lvert\mathcal{S}^{+}\rvert$	778 419	308 004	142 221	24 257	13 765	14 157
$\lvert E_L\rvert$	2 298 358	1 038 032	393 256	60 154	45 882	42 422
$\lvert V^{\delta}\rvert$	8 873 182	3 802 306	1 559 118	242 249	191 685	220 188
$\lvert E^{\delta}\rvert$	23 949 850	11 015 006	4 206 427	645 831	886 138	601 572
$\lvert V^{\varepsilon}\rvert$	204 806 646	58 267 484	38 370 666	9 427 272	8 841 424	6 342 708
$\lvert E^{\varepsilon}\rvert$	572 461 090	187 336 609	105 731 613	21 775 337	80 346 645	16 908 445

(a) Europe

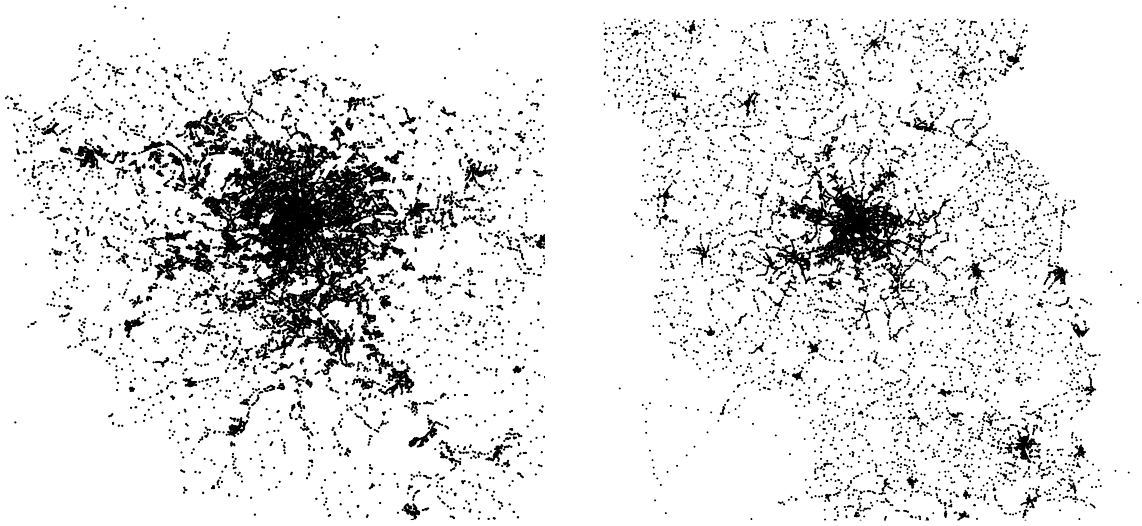


(b) Germany

(c) Great Britain

(d) Switzerland

Figure 5.1: Visualization of the large datasets: Europe, Germany, Great Britain and Switzerland.



(a) Paris (b) Berlin

Figure 5.2: Visualization of the metropolitan datasets: Paris and Berlin.

5.2 Preparation

Many GTFS datasets contain apparently unnecessary stops that are not served at all. We removed stops without any event. Some of these are stop areas, which are intended to represent a number of stops as one area, or stops that can be served by rail replacement services if necessary.

In addition, we remove all trips that do not operate within the specified period, namely two consecutive weekdays. Special trips, e.g., ones that that only operate on weekends or on public holidays, are no longer included.

To group all trips into FIFO lines, we discard the information about lines from the GTFS files, as these do not necessarily fulfill the FIFO property (e.g., a line in the GTFS sense may identify both the inbound and the outbound direction). There is a polynomial-time algorithm [38] that groups a set of trips into an optimal, i.e., smallest, set of lines, but a simple greedy algorithm also provides the same solution as the optimal algorithm on all our networks; however this algorithm is much faster.

All trips are grouped according to their stop sequence. Within each group, the trips are sorted lexicographically by departure times. Subsequently, each group is partitioned greedily into FIFO lines, where the FIFO property must hold. The trips in each group are processed in sorted order: for each trip, it is checked whether it can be added to an existing line (of this group) without violating the FIFO property. If this is the case, the trip is added to that line; otherwise, a new line is created.

We create the transitive closure of all footpaths, using Dijkstra's algorithm [14] starting from each stop $p \in \mathcal{S}$. If the algorithm finds a new stop $q \in \mathcal{S}$, it creates the edge (p, q), if not already present, and updates its weight to the correct shortest path length.

The TB data structures used are implemented as in the TB paper [1]. All lines from the datasets have less than 256 stops, i.e., it takes 8 bits to identify the stop index along a line. We take advantage of this to store an IBE compactly in a 32-bit number, namely by using the lower 8 bits for the stop index and the remaining 24 bits for the trip ID. This allows for very low memory overhead and enables efficient sorting. Cell IDs are unsigned 16-bit integers, and the rank of a transfer is an unsigned 8-bit integer. It is important to keep these two data types as small as possible to allow loading many transfer ranks per cache line, thus reducing cache misses and loading overhead.

All IDs of stops, trips, lines, and events are numbered consecutively starting from 0. We distinguish between static and dynamic graphs, where dynamic graphs allow edges to be added or removed. Dynamic graphs are typically implemented as edge lists or as vectors of vectors, where each vertex stores its outgoing (and sometimes incoming) edges in a separate vector.

The static graph is implemented using a forward-star (adjacency array) representation. In this structure, the graph consists of two main arrays: `adj` and `head`. The array `adj` has size $|V| + 1$, and at position $v \in V$ (where vertex IDs are used directly), it contains an index

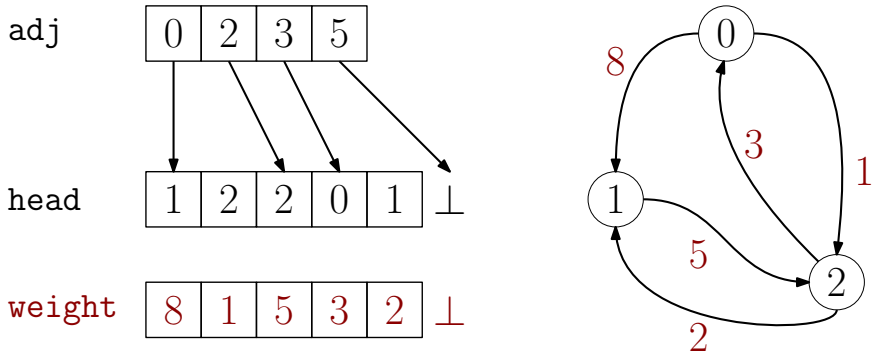


Figure 5.3: Example graph (right) with three vertices (numbered 0 to 2) and five edges, shown in forward-star representation on the left. The `adj` array points to ranges in the `head` array, which lists target vertices of each edge. Corresponding edge weights are shown in red. For instance, vertex 0 has edges to vertices 1 and 2 with weights 8 and 1.

pointing to the first edge starting from v in the `head` array. The `head` array, which has size $|E|$, stores the IDs of the target vertices of all edges. Thus, the outgoing neighbors of a vertex v are given by accessing $\texttt{head}[\texttt{adj}[v]]$ up to $\texttt{head}[\texttt{adj}[v+1]-1]$. Additional edge information, such as `weight` or `traveltime`, is stored in separate arrays of the same length as `head` and accessed in the same way. This structure is compact and cache-efficient, but does not support dynamic modifications to the graph (besides edge-weight updates) without rebuilding the arrays. See Figure 5.3 for an illustrative example.

5.3 Graph Partitioner

We evaluate T-REX on the basis of several graph partitioners that use different approaches to partition graphs. The first partitioner is a black-box partitioner Mt-KaHyPar [33, 34, 35](Multi-Threaded Karlsruhe Hypergraph Partitioner), an open-source framework[8] based on the multilevel paradigm to partition arbitrary (hyper)graphs. The multilevel approach is an effective heuristic for balanced partitioning. It involves three phases. First, sets of vertices are iteratively contracted during the coarsening phase to create smaller (hyper)graphs. Then, an initial partition is made, followed by iteratively refining the partition as the contractions are reversed. In our results, the option `cut`, i.e., to minimize the sum of all *cross edges*, delivered the best results as an objective function. It should be mentioned that in initial experiments KaHIP [39], which is also an open-source multilevel blackbox partitioner[9], delivered similarly good results, but with significantly longer runtimes.

PUNCH [40] (Partitioning Using Natural Cut Heuristics) delivers, as far as we know, the best results for CRP [8], as this algorithm is designed in a way to exploit “natural” cuts

[8] `https://github.com/kahypar/mt-kahypar`
[9] `https://github.com/KaHIP/KaHIP`

in road networks, such as rivers or mountain ranges. Since there is no publicly available implementation of PUNCH, we tested Buffoon [41], a KaHIP-based algorithm that exploits the same "local", natural cuts. Unfortunately, Buffoon did not achieve good results, which was to be expected. For example, while a bridge in the road network is a good candidate for a cut edge, the bridge may be heavily used by trains and therefore not an adequate cut edge.

Simpler partitioning methods were also tested experimentally, but were quickly discarded due to bad customization times and query performance. Note that both methods only use the coordinates of the stops, but do not use any information from the compact layout graph. The implementations can be found here [10].

The first algorithm performs hierarchical k-means clustering, where it recursively partitions a set of points into clusters by first applying 2-means clustering. It continues to split each resulting cluster into smaller sub-clusters until 2^ℓ clusters are formed. Note that k-means clustering was also used in Scalable Transfer Patterns [18], in combination with a post-processing step to merge some clusters together in order to reduce the number of border stops. The second method, a recursive coordinate splitter, is likewise hierarchical but operates analogously to a 2-d tree. Alternating per level, the stops to be partitioned are sorted by latitude or by longitude and then halved into two cells. Both halves are then partitioned in the same way.

All partitions used in the following experiments were computed with Mt-KaHyPar. It took only seconds to partition even Europe into 2^{16} cells. More precisely, we use the preset `highest_quality` and as the initial partitioner we use the option `flow_cutter`. The exact command can be found in the code repository[11].

As recursive bipartitioning is performed top-down, imbalance can accumulate across levels. To prevent this, Mt-KaHyPar adaptively reduces the allowed imbalance ε at each step to ensure that the final 2^ℓ-way partition respects the global imbalance bound. This is done by computing a smaller local imbalance $\varepsilon'(l) \leq \varepsilon$ for each level $l \in [1, \ell]$; see [42] for more details. For our experiments, we modified Mt-KaHyPar[12] to set the imbalance of the topmost bipartition to a high value, i.e., $\varepsilon'(1) = 10.0 = 1000\%$, while using the given value of ε for all subsequent levels. This relaxed initial cut allowed more flexibility early on and led to better overall results.

5.4 Customization

Figure 5.4 plots the performance of the customization on all networks. Precomputation time primarily depends on the number of IBEs, which in the worst case increases exponentially with the number of levels. More levels result in more cells, and therefore more IBEs. Allowing a more imbalanced partition reduces the cut size. However, this may also lead to

[10] `https://github.com/PatrickSteil/CoordRecSplitter`
[11] `https://github.com/PatrickSteil/TREX`
[12] `https://github.com/PatrickSteil/mt-kahypar`

some cells being much larger or denser than others. These cells are more difficult to process, as the search within them requires significantly more work. This explains why, especially with a small number of levels, the precomputation times can vary widely for different imbalances (see e.g., Figure 5.4d; Switzerland instance with 6 levels and an imbalance of $100\,\%$).

For each network, we evaluated four imbalance values per level ($25\,\%$, $50\,\%$, $75\,\%$ and $100\,\%$). In subsequent results, such as query performance or customization time, the configuration achieving the best performance among these four imbalances is used, unless explicitly stated otherwise.

The customization is fast: even on a single core, the Germany instance (for 9 levels with an imbalance of $25\,\%$ on the Epyc machine) requires only 715 seconds, i.e., $11{:}55$ mm:ss. As we will show in the next section, the simple parallelization scheme is very effective.

Note that the customization considered here is "unbounded" in the sense that it covers the complete service period of the respective datasets, i.e., at least $48\,\mathrm{h}$. This means that all IBEs in this time frame are processed. For real-world use, one would not customize complete timetables, but only certain time periods and even possibly only local areas (e.g., metropolitan regions). This significantly reduces customization time. For instance, on the Germany network, if only the IBEs departing between $11{:}00{:}00$ and $13{:}00{:}00$ (the two-hour period with the highest trip density) are considered, customization in the fastest configuration (i.e., with 9 levels and an imbalance of 25%) is completed in just 1.8 seconds using 128 threads.

One important thing that makes customization much faster is the relaxation of "relevant" transfers per level. While processing an IBE at level i, the customization only has to relax transfers that have a rank of exactly i, since the customization works bottom-up. This means that unimportant transfers are no longer explored. Without this optimization, on Germany in the fastest configuration, it no longer takes 13 seconds, but around 39 seconds. Without this simple optimization, the runtime does not decrease per level, as no part of the search space can be pruned.

In the appendix, Figure A.1 shows the number of IBEs per level for each network for the best performing configuration, along with a more detailed breakdown of the customization times per level.

5.4.1 Efficiency

Let p denote the number of processors and $T(p)$ the execution time using p processors. The *speedup* $S(p)$ is defined as the ratio of the best known sequential to parallel execution time, i.e., $S(p) = {}^{T_{\text{seq}}}/_{T(p)}$. The *efficiency* $E(p)$ of a parallel algorithm is an important measure of its performance. It quantifies how well additional processors are utilized to accelerate the computation and is defined as $E(p) = {}^{S(p)}/_{p}$. Ideally, the efficiency is 1, i.e., the work was divided equally among all processors, and there was no overhead due to communication or

synchronization. In other words, the parallel execution time is exactly $1/p$ times the sequential execution time. However, achieving an efficiency of 1 is rarely possible in practice due to factors like communication overhead, load imbalance, and inherent sequential components.

See Figure 5.5 for the different efficiency measures of the customization algorithm, and its detailed efficiency *per level*. The used dataset is the Germany network, using 9 levels and an imbalance of $25\,\%$. Please note $T_{\text{seq}} \equiv T(1)$.

For up to 64 threads, the parallelization scheme maintains an efficiency greater than 0.6, with efficiency around 0.8 for up to 16 threads; cf. Figure 5.5a. This indicates that the customization is highly parallelizable, meaning the straightforward division of IBEs per level scales well with an increasing number of computing cores. However, we observe that when using 128 threads, the efficiency drops, likely because the `Epyc` machine becomes memory-bound at this point, meaning that memory access speed is the limiting factor in performance. Similar effects have been observed on this machine before, e.g., see [5].

We also report the efficiency per level in Figure 5.5b. This is similar to the total efficiency in Figure 5.5a, but clearly shows that higher levels can be processed much more efficiently. Higher levels have fewer cells, and therefore fewer IBEs, which shows that the distribution of tasks among the threads for smaller levels costs more and more time.

5.4.2 Transfer Distribution

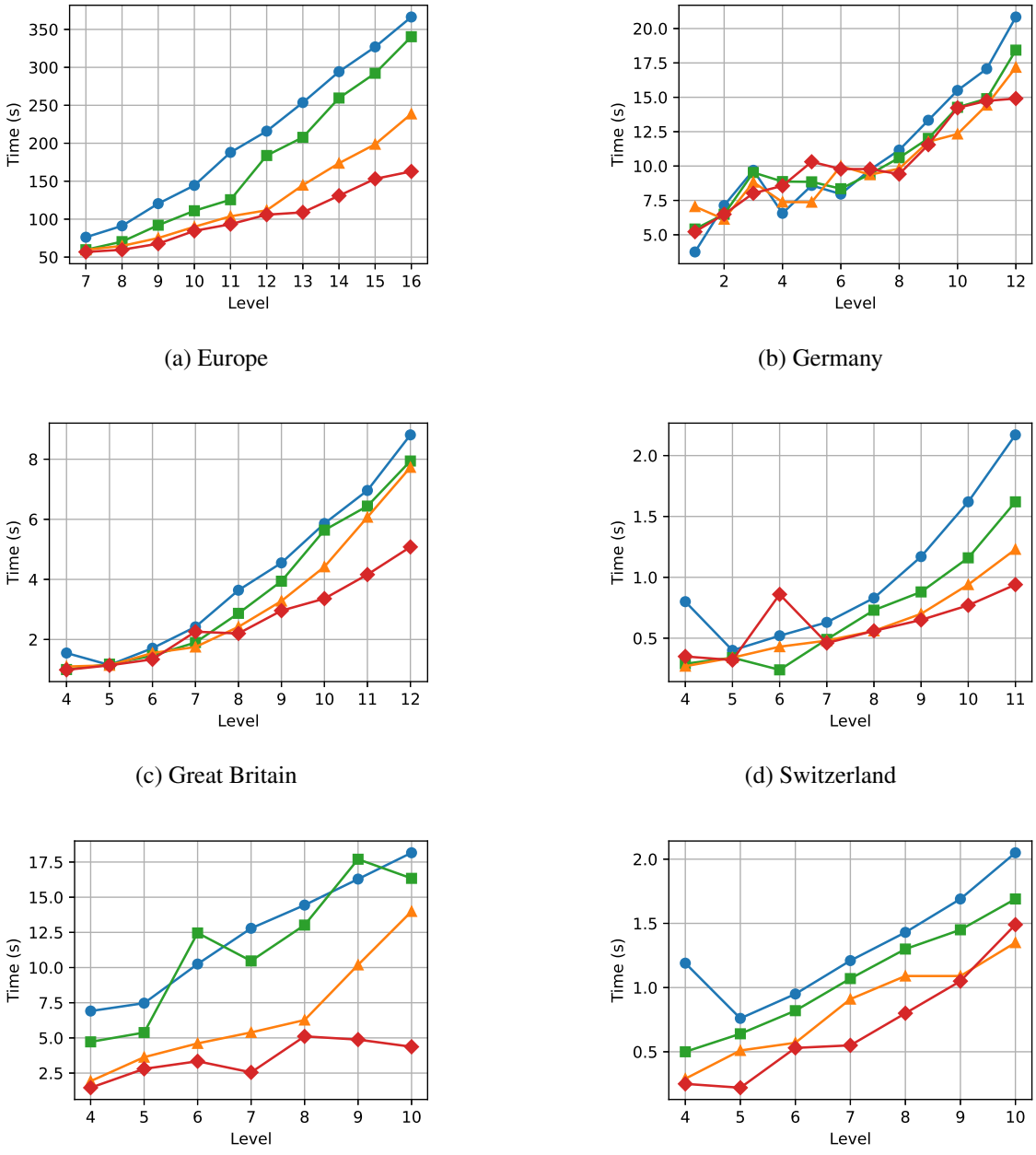
Another interesting aspect is the distribution of ranks. The authors of FLASH-TB have also examined a similar measure, namely the distribution of flags. Given a k-partition, each transfer is assigned k boolean flags, where the i-th flag is set to true if this transfer is part of an optimal journey to a stop in the i-th cell. They observed that almost $80\,\%$ of the transfers on the country networks have a maximum of two flags set to true (see [5], Figure 6). Since the majority of transfers are apparently only relevant for answering local queries, we expect a similar picture here.

Indeed, our observations largely confirm this expectation. In the country networks, a significant fraction of transfers have low ranks, indicating that most transfers are primarily relevant for local journeys. In the following, we distinguish between the country networks and the metropolitan regions; cf. Figure 5.6.

For the large networks, it is clear that there are significantly more transfers with a lower rank, which was to be expected. This is particularly pronounced in Germany, where almost $85\,\%$ of all transfers have a rank of zero (cf. Figure 5.6b). Switzerland, on the other hand, has a much flatter distribution (cf. Figure 5.6d). In the other three large networks, $> 50\,\%$ of all transfers have a rank of zero; on Switzerland, just under $50\,\%$ of all transfers have a rank of less than two. This is also reflected in the query performance. Compared to the other nationwide networks, T-REX achieves a lower speedup on Switzerland.

The metropolitan regions, on the other hand, show that there is (almost) no “hierarchy”, and thus no real distinction can be made between “local” and “global”. The distributions of

Paris (Figure 5.6e) and Berlin (Figure 5.6f) are very flat; in Paris only just under half of all transfers have a rank lower than five. These observations further confirm that metropolitan regions are inherently different from country- or continent-sized networks, whether due to their high *density* of trips and footpaths or their lack of hierarchical structure, i.e., the absence of interplay between local and long-distance transport.



(a) Europe

(b) Germany

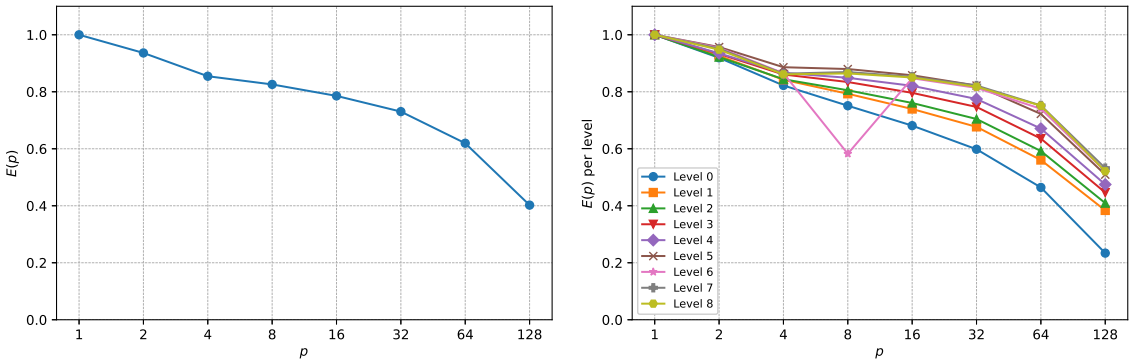
(c) Great Britain

(d) Switzerland

(e) Paris

(f) Berlin

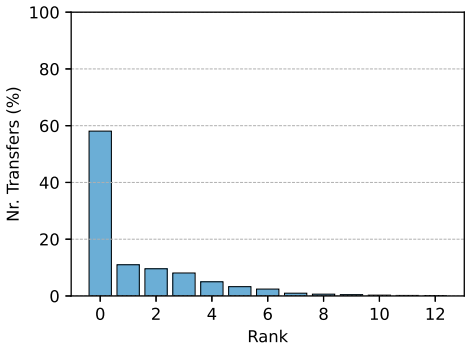
Figure 5.4: The precomputation time of the customization per network is shown, differentiated by level and imbalance. Each imbalance is represented with a unique symbol and color for improved readability: a red diamond (♦) for 100%, an orange triangle (▲) for 75%, a green square (■) for 50%, and a blue circle (●) for 25%. All experiments were conducted on the Epyc machine. The Europe instance was run with 64 threads, while all other networks used 128 threads.



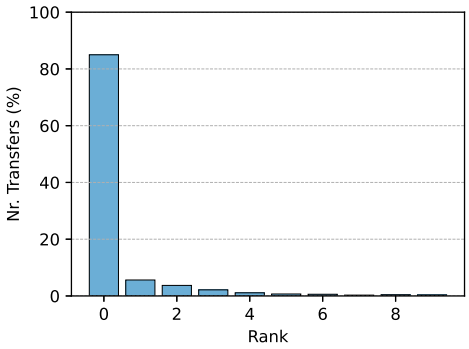
(a) Total Efficiency

(b) Efficiency per level

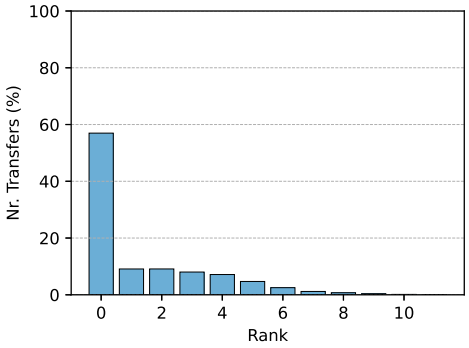
Figure 5.5: Shown are different efficiency plots for the customization on the Germany dataset for 9 levels with an imbalance of 25 % on the Epyc machine.



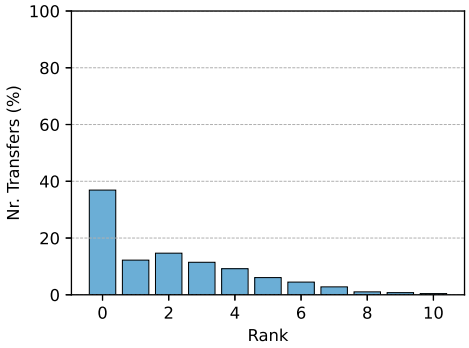
(a) Europe



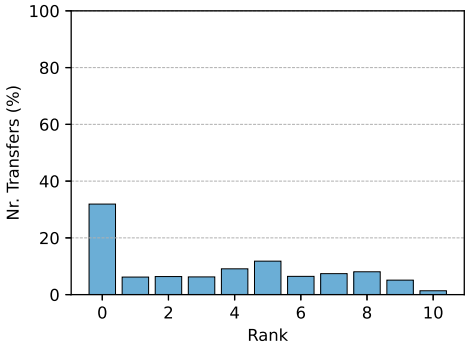
(b) Germany



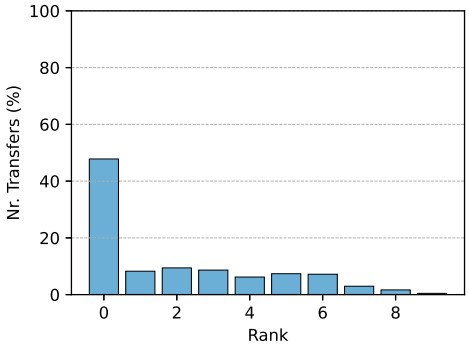
(c) Great Britain



(d) Switzerland



(e) Paris



(f) Berlin

Figure 5.6: The figures show the distribution of transfers assigned to each rank. For every network, we choose the distribution corresponding to the fastest partition (cf. Table 5.5 and Table 5.6). The y-axis is fixed from 0 to 100 % to allow an easy comparison between the histograms.

5.5 Query

We evaluate T-REX, TB, RAPTOR, CSA, ACSA, TED and TDD on all networks, while FLASH-TB and TB-CST are evaluated only on a subset of networks due to the high precomputation time and memory requirements of the largest networks. The original code for CSA and ACSA used in [10] was provided to us by the authors. The TED implementation follows the PTL paper [16], which *contracts* the departure vertices, as they all have an outdegree of one.

We implemented TDD using the more detailed modeled presented in Section 3.3.1. Furthermore, the performance results of FLASH-TB and TB-CST [5] for Germany and Paris are directly comparable to our results, as both networks were derived from the same sources and evaluated on the same machines using the same setup. The small difference in the size of the Germany network is due to the fact that different days are extracted.

5.5.1 Methodology

We evaluate the query performance in three different ways:

In the first method, which we refer to it as *fixed departure time query*, the source and target stops are chosen uniformly at random, and a departure time is chosen uniformly at random between $00{:}00$ and $23{:}59$ on the first day of the service period. We perform a total of $10\,000$ fixed departure time queries and report average metrics such as query time and the number of scanned trips.

In the second method, the *fixed departure time geo-rank query*, the source stop is selected uniformly at random, and all remaining stops are sorted according to their geographical distance from the source. Subsequently, for each geo-rank r, a fixed departure time query is performed to the 2^r-th closest stop, where r denotes the geo-rank. The departure time is again chosen uniformly at random within the first day. The aim of this is to evaluate the fixed departure time query in terms of geographical proximity, i.e., to get a clearer impression of how fast "local" or "global" queries can be answered. We select $10\,000$ random source stops and perform one fixed departure time query per geo-rank r. Unlike the other experiments, we visualize the resulting distributions using whisker plots rather than reporting averages.

The third method *profile query* evaluates the performance of profile queries by again choosing the source and target uniformly at random, and setting the time range as the entire first day (i.e., $[\tau_{\text{start}}, \tau_{\text{end}}] = [00{:}00, 24{:}00]$). We perform a total of $1\,000$ queries and report average metrics such as query time and the number of scanned trips.

The query times for fixed departure time queries do not vary significantly depending on the departure time (within the first day), at least for TB- and RAPTOR-based algorithms. This is because these algorithms, for all reachable lines, only scan the next departing trip. In other words, a line is no longer considered once its last reachable trip has been processed. However, since we consider two consecutive days in our networks, nearly every line has

Table 5.2: Comparison between T-REX and TB. Average fixed departure query times for 10 000 uniformly random source–target stop pairs. Shown are total preprocessing times and the chosen parameters k and imbalance. Preprocessing for T-REX include the time to generate the transfers. See Subsection 5.5.1 for the experimental setup.

Network	Algorithm	Query [μs]	Prepro. [mm:ss]	k	Imbalance [%]
Europe	TB	222 834	05:26	-	-
	T-REX	18 009	09:01	4 096	25
Germany	TB	71 030	00:45	-	-
	T-REX	8 331	00:58	512	25
Great Britain	TB	17 477	00:24	-	-
	T-REX	2 293	00:30	2 048	50
Switzerland	TB	4 605	00:04	-	-
	T-REX	924	00:06	1 024	25
Paris	TB	3 746	00:07	-	-
	T-REX	2 018	00:18	1 024	25
Berlin	TB	3 055	00:04	-	-
	T-REX	1 046	00:05	512	50

trips on the following day. Note that all three methodologies are common practice (see e.g., [5, 7, 10]).

We limit TB, RAPTOR and T-REX to 16 rounds, whereas CSA, ACSA, TED, TDD and PTL only compute the fastest journey. We did not measure the time to perform journey unpacking. The data for TB-CST and FLASH-TB for Germany and Paris are taken from [5]. For TB-CST, we take the fastest query times of either the split-tree (denoted by TB-CST$^{\text{s}}$) or prefix-tree (denoted by TB-CST$^{\text{p}}$) variant. All preprocessing was performed on the `Epyc` machine with 128 threads (except Europe, where only 64 threads where used for T-REX and ACSA due to high memory usage), whereas the queries were evaluated on the `Xeon` machine. The preprocessing times for all algorithms based on TB include the preprocessing times to generate the transfers.

5.5.2 Fixed Departure Time Queries

T-REX achieves a speedup on all networks compared to the baseline TB, cf. Table 5.2. On Europe, T-REX can answer a random query in 18 ms, making it fast enough for real-world interactive applications. TB is an order of magnitude slower at 222.8 ms, taking almost a quarter of a second. On Germany, T-REX achieves a speedup of around 8.5, with a query

performance of $8.3\,\mathrm{ms}$. For Great Britain, a similar picture emerges; a speedup of 7.6 and a query time of $2.3\,\mathrm{ms}$. On such large-scale networks, which connect dense, local structures (metropolitan regions) with long-distance traffic, the algorithm achieves a good speedup of 7–12. In contrast, Switzerland shows a more moderate speedup of about five, yet still yields sub-millisecond query times ($0.92\,\mathrm{ms}$).

For metropolitan areas, almost any modern algorithm is fast enough to be used in real-world applications (as will be shown in Section 5.6). Nevertheless, T-REX still achieves a small speedup over TB on both Berlin and Paris. Paris, however, performs worse than Berlin, which is most likely due to the high footpath density. As footpaths are contracted, the compact layout graph becomes denser and smaller, leaving little room for good partitions and thereby limiting the achievable query speedup.

In Table 5.3, we also evaluate how different query metrics such as query time, scanned trips and relaxed transfers are affected by the number of levels. For completeness, the same experiments were performed for ACSA as well. See Tables A.1, A.2 and A.3 for a similar overview on the other networks.

Initially, adding levels drastically reduces these metrics, leading to faster queries. However, the speedup plateaus at nine levels; any additional level provides only marginal improvements. This behavior correlates closely with the number of scanned trips and relaxed transfers, both of which decrease significantly as the number of levels increases. Initially, adding levels leads to drastic reductions in these metrics, resulting in faster queries. For instance, transitioning from zero to four levels decreases the number of scanned trips by almost $72\,\%$ and relaxed transfers by around $70\,\%$, leading to around $70\,\%$ reduction in query time. Beyond level nine, however, the number of scanned trips and relaxed transfers decrease only slightly, which explains why improvements in query time are negligible.

The original implementation of the TB algorithm by Witt [1] is highly optimized and incorporates assumptions about the network's size and metric properties to achieve optimal performance. In particular, the number of bits used to represent certain data elements (such as events, transfers or custom structures used during the query) is minimized to the smallest possible length. This is also reflected in the query performance compared to our implementation on the same networks. For random queries, the TB implementation by Witt needs $32.59\,\mathrm{ms}$ on Germany, $6.69\,\mathrm{ms}$ on Great Britain $1.75\,\mathrm{ms}$ on Switzerland, $1.59\,\mathrm{ms}$ on Paris and $1.34\,\mathrm{ms}$ on Berlin. However, due to the hard-coded assumptions and bit-width limitations, the European dataset cannot be correctly loaded, and TB therefore cannot be executed on this instance. Nevertheless, these results demonstrate that tailoring an implementation to the characteristics of a specific dataset can yield substantial performance gains and thus faster customization. Since the speedup of T-REX over TB primarily results from its restriction of the search space, it remains largely independent of the degree of fine-tuning applied to the underlying TB implementation.

T-REX is designed to efficiently answer stop-to-stop queries. Consequently, many-to-many queries must be decomposed into multiple pairwise stop-to-stop queries. While these

Table 5.3: Listed are average query metrics of T-REX and ACSA per level on the Germany network. For T-REX, we report the number of scanned trips, relaxed transfers, query time and preprocessing. 10 000 random queries were evaluated using the fixed departure time queries method. For level zero, we report the metrics for TB and CSA. Preprocessing for T-REX does not include the time to generate the transfers. Query times are shown as $[\mu s]$, preprocessing as $[\mathrm{mm{:}ss}]$. For every level, we choose the imbalance that led to the best query performance.

Levels	T-REX				ACSA	
	Scan. Trips	Rel. Transfers	Query	Prepro.	Query	Prepro.
0	316 797	4 675 832	70 640	–	82 261	–
1	254 773	3 826 496	61 128	**00:07**	138 379	00:23
2	176 743	2 754 992	43 275	**00:07**	119 222	00:29
3	123 162	2 026 925	31 365	00:08	95 950	00:26
4	86 672	1 492 093	23 111	**00:07**	75 430	00:16
5	46 960	945 748	14 669	00:09	51.508	00:12
6	32 575	742 908	11 809	00:08	37 961	**00:09**
7	23 206	618 404	9 764	00:10	25 310	**00:09**
8	19 248	564 201	8 879	00:11	20 025	**00:09**
9	17 341	**531 454**	**8 331**	00:13	**18 333**	00:10
10	17 624	540 250	8 363	00:12	19 082	00:12
11	18 089	547 592	8 891	00:14	21 013	00:17
12	**17 125**	544 389	8 332	00:17	20 087	00:24

individual queries are answered very efficiently, we expect that for aggregate query types such as one-to-all or all-to-all, algorithms like TB or RAPTOR would perform better.

In practice, however, many-to-many queries occur quite often, since given a start and destination location (such as GPS positions), the surrounding stations are first collected and optimal journeys are computed between the two sets. For the simple case where all source stops are located in the same cell, as well as all the target stops, T-REX can be used directly. However, as soon as one source or destination stop is located in another cell, the *source-target* pruning via LCL is no longer viable. This could be an interesting question for future research.

5.5.3 Fixed Departure Time Geo-Rank Queries

As is common practice, we represent the running times for geo-rank queries using whisker plots, which extend from the first to the third quartile, with a line showing the median and, if present, potential outliers.

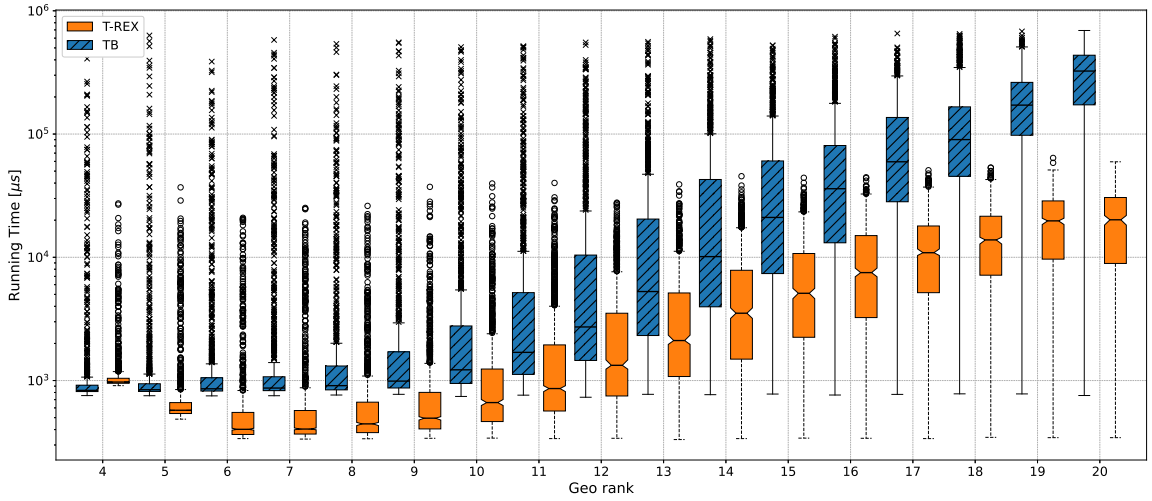


Figure 5.7: The average query times $[\mu s]$ to the 2^i-th closest stop are shown, where i is the geo-rank, collected over $10\,000$ random queries. The network is the European dataset, and TB (blue) and T-REX (orange) are compared. The "whiskers" extend the box, and represent the $1.5\times$ values of of the first and third quartile, respectively. Note that the y-axis is in logarithmic scale.

We evaluated the geo-rank queries on the Europe dataset using the configuration with the number of levels that yielded the fastest fixed departure time queries; cf. Figure 5.7. T-REX outperforms TB in almost all cases. The only exception is for very local requests (with geo-rank four), where both algorithms require just under $1\,\mathrm{ms}$. The reason for the slightly worse performance of T-REX is the additional overhead per transfer to perform the LCL check. For such local queries, very few transfers are relaxed, so the overhead of pruning the search space is not justified. For geo-ranks between six and nine, T-REX achieves runtimes of 0.3 and $0.5\,\mathrm{ms}$. By comparison, TB needs 0.8 to $1\,\mathrm{ms}$ for a geo-rank of four up to nine. For larger geo-ranks, the speedup of T-REX over TB increases, which was to be expected. Even outlier queries are processed more efficiently by T-REX, requiring only 55 to $60\,\mathrm{ms}$; an order of magnitude faster than TB.

The query times reported in Tables 5.5 and 5.6 are average runtimes for random source and target stops, and which are far apart on average. The approximate $18\,\mathrm{ms}$ are obtained for queries with rank 19, and therefore do not exactly reflect real-world usage. The queries that are relevant in practice (i.e., many local, a few global) are answered very quickly.

5.5.4 Profile Queries

We ran the $1\,000$ random profile queries for each network using for every network the fastest configuration of levels and imbalance; cf. Table 5.4.

Table 5.4: The table shows the average query metrics of 1 000 random profile queries, such as query time, the number of trips scanned, relaxed transfers and the number of journeys found.

Network	Algorithm	Scan. Trips	Rel. Transfer	Journeys	Query [μs]	k
Europe	TB	3 544 302	69 398 487	12.60	1 298 820	-
	T-REX	99 419	4 624 444	12.60	99 639	4 096
Germany	TB	1 242 125	17 510 793	10.78	334 259	-
	T-REX	70 091	1 941 681	10.78	43 850	1 024
Great Britain	TB	358 507	6 579 673	11.02	100 993	-
	T-REX	22 450	690 214	11.02	13 373	2 048
Switzerland	TB	216 323	2 085 029	24.23	37 862	-
	T-REX	26 326	349 671	24.23	6 988	1 024
Paris	TB	181 380	4 772 478	32.57	42 162	-
	T-REX	70 222	2 358 960	32.57	22 793	1 024
Berlin	TB	160 394	1 848 622	29.44	26 967	-
	T-REX	33 591	573 637	29.44	9 392	512

In T-REX and TB, profile queries are solved iteratively by running multiple fixed departure time queries, for each departure time within the specified interval. As a result, the data structures required for journey reconstruction are overwritten before each iteration. To preserve all journeys found by the algorithm, it is therefore necessary to unpack and store the results of each query before proceeding to the next; otherwise, the information would be lost. We measured the time required by both algorithms to perform journey unpacking.

For the European instance, unpacking took $100\,\mu$s, Germany $64\,\mu$s, Great Britain $113\,\mu$s, Switzerland $90\,\mu$s, Paris $300\,\mu$s and for Berlin $115\,\mu$s. Journey unpacking is marginally faster for T-REX (at most 2 μs faster) as the overall queue is smaller and there are therefore fewer cache misses when following the parent pointer. However, the times correlate with the number of journeys found and are negligible compared to the query times.

Once again, we see that T-REX can very effectively limit the search space, achieving a significant speedup compared to the TB baseline. On all networks, the speedup of the profile query is also roughly equivalent to that of the fixed departure time query. The speedup is particularly high on Europe, with a factor of 13 and an average runtime of just under $100\,\mathrm{ms}$.

5.6 Comparison With Other Algorithms

In this section, we compare the performance of T-REX in detail with other state-of-the-art algorithms, such as ACSA, (Scalable) Transfer Patterns, FLASH-TB, TB-CST, and labeling algorithms on time-expanded graphs, in detail. See Tables 5.5 and 5.6.

5.6.1 ACSA

The most similar and most fairly comparable algorithm is ACSA, of which we have the original code and were therefore able to evaluate all algorithms on the same networks and on the same machines. We want to remind that CSA and ACSA only solve the simpler earliest arrival problem, whereas T-REX calculates the Pareto-optimal journeys in terms of arrival time and number of transfers.

We evaluate ACSA on the same partitions as T-REX, i.e., we only evaluate nested bipartitions. This means that the partitions are not computed using the original code, but with the help of Mt-KaHyPar. The original publication [10] showed that ACSA achieved the best query performance with a nested bipartition.

For an overview of the comparison of both customizations, see Figure 5.8; and for query performance, see Table 5.5 and Table 5.6.

Customization

Both algorithms are evaluated using the same number of threads, i.e., Berlin, Paris, Switzerland, Great Britain and Germany with 128, and Europe with 64 (cf. Figure 5.8). This is because T-REX preallocates an array for each thread in the size of the number of events. This becomes too large with 128 threads, whereas ACSA has a much smaller memory footprint for each thread used, and can also run on Europe with 128 threads.

On the Europe network, ACSA outperforms T-REX in terms of customization time; being up to $60\,\%$ faster. However, as the level increases, this advantage decreases significantly, and at the highest level, T-REX becomes slightly faster. On the Germany network, ACSA is again faster for levels between 6 and 10, achieving up to $45\,\%$ faster customization.

For all other networks, T-REX consistently performs better, sometimes requiring only 20–40 % customization time compare to ACSA. Overall, as the number of levels increases, the customization times of both algorithms tend to converge.

Fixed Departure Time Query

In terms of query times, T-REX is faster than ACSA (cf. Table 5.5 and 5.6). On Europe, both algorithms achieve a speedup of about one order of magnitude compared to their baseline (ACSA with $38.51\,\mathrm{ms}$, T-REX $18.01\,\mathrm{ms}$). On the Germany network, ACSA achieves a

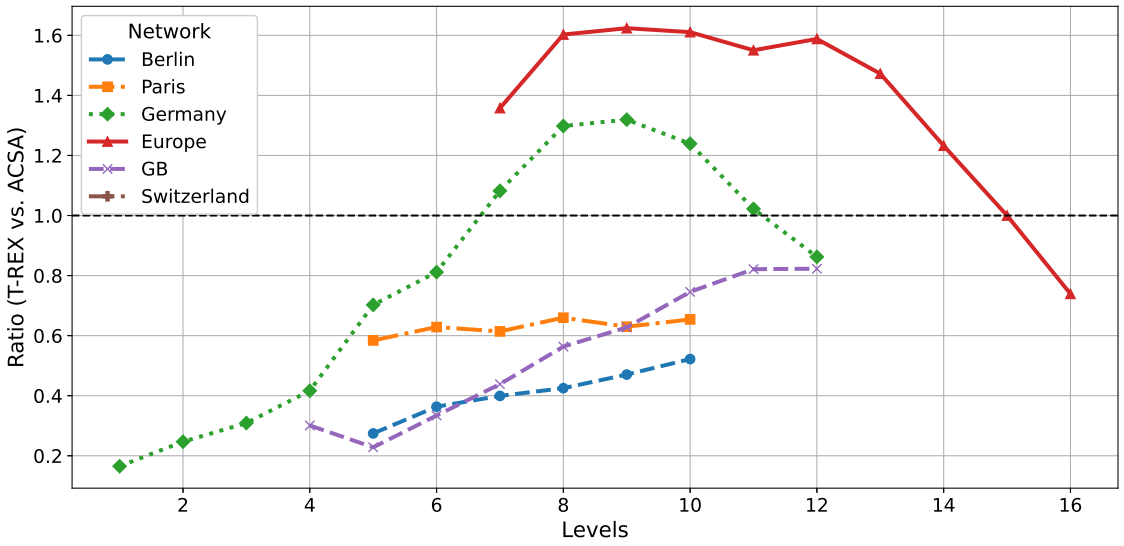


Figure 5.8: Ratio of T-REX to ACSA customization times for the different datasets and levels using the same partitions (with 25 % imbalance). A value $x > 1$ means T-REX takes x times longer than ACSA, while a value below 1 means T-REX is faster by a factor of $1/x$.

speedup of 4.25 compared to CSA, while T-REX reaches just under 8 compared to TB. A similar discrepancy can also be seen on the Great Britain instance; ACSA with 4.2, and T-REX 6.8. On Switzerland, the speedup of both algorithms is lower. ACSA achieves a speedup of 1.88, T-REX 4.4 (and has query times of just under 1 ms).

A different picture emerges for the metropolitan networks (cf. Table 5.6). Here, ACSA is slower than the baseline CSA, while T-REX remains slightly faster than TB, but the speedup is rather low. As expected, the effectiveness of the multilevel approach is closely tied to the size and structure of the network; explaining why Europe and Germany benefit more than Paris and Berlin.

A similar trend was observed in the comparison between FLASH-TB and TB-CST. As the authors of FLASH-TB [5] have shown, FLASH-TB benefits from not having to set up any additional data structures at query time, unlike TB-CST or TP. The same applies here: The runtime of ACSA on metropolitan regions is dominated by the merging of the connections; the mere scanning of the sorted connections is very fast. T-REX does not need to build any additional data structure during the query, it evaluates whether a transfer should be relaxed or not on the spot; as does FLASH-TB.

In conclusion, both algorithms demonstrate strong performance, and the multilevel partitioning approach combined with TB proves highly effective. Although T-REX tackles a more complex problem, its queries are faster than those of ACSA. With modern partitioners such

as Mt-KaHyPar, high-quality partitions can be generated within seconds, benefiting both T-REX and ACSA.

5.6.2 Graph-Based Algorithms

In this section, we compare the performance of graph-based algorithms. First, we discuss the Dijkstra algorithms TED and TDD, and then we take a look at PTL.

Time-Expanded and Time-Dependent Dijkstra

See Tables 5.5 and 5.6. TED is the slowest algorithm, which was to be expected. TDD is faster than TED by a factor between three and six. Surprisingly, TDD on Paris is faster than TED by a factor of 13, which is due to the fact that the many footpaths are not represented multiple times in the underlying graph, but only once (as an edge from stop to stop). RAPTOR beats TED and TDD, even though RAPTOR computes Pareto-optimal journeys. There is a clear trend that array-based algorithms perform better than graph-based approaches. This is plausible, as entire lines or trips are stored consecutively in memory and can therefore be iterated over their stops in a sweep (e.g., to update arrival times). This observation is consistent with previous findings in the literature [2, 22], which report similar performance advantages for array-based algorithms over graph-based ones. Dijkstra's algorithm [14] suffers from poor cache efficiency, partly due to cache-unfriendly priority queue operations, in particular the `decrease-key` operation [43]. Another contributing factor is the scattered nature of memory accesses, since data associated with neighboring vertices (like minimal arrival time or parent pointer) are usually not stored contiguously in memory, leading to poor spatial locality during edge relaxation.

Labeling Algorithms

We have also implemented PTL [16], a Hub Labeling-based algorithm. Given precomputed hub labels, PTL can answer queries in a few microseconds. As a precomputation algorithm, the authors use RXL [44], a state-of-the-art Hub Labeling algorithm. Unfortunately, we do not have access to the original implementation of RXL, and although a public implementation[13] delivers good results in terms of label sizes and query performance, the precomputation time is not comparable to the original publication. For example, calculating hub labels for non-Pareto optimal queries on our Berlin instance took more than $> 29\,\mathrm{h}$, whereas the precomputations in the original paper were completed in less than one hour. We also tried a different Hub Labeling algorithms (see Appendix A.2), but the resulting labels (and thus the total memory consumption) were significantly higher than in the original paper [16] and hence not comparable.

[13] `https://github.com/lviennot/hub-Labeling`

Query times for PTL are significantly faster than those of T-REX, but this comes at the cost of higher precomputation time and very high memory consumption. For instance, the authors report that for Pareto-optimal queries on the Switzerland instance, memory consumption is around $13\,\mathrm{GB}$ and precomputation takes $61\,\mathrm{h}$, resulting in an average query time of $21.7\,\mu\mathrm{s}$. Because of these trade-offs, a direct comparison provides only limited insight. To address this, we consider a variant of Hub Labeling approaches: Hybrid Labeling, which offers a compromise between precomputation time, memory consumption, and query performance.

The underlying idea is to precompute labels as well, but instead of just computing the intersection of forward and backward labels at query time (the standard Hub Labeling approach), these labels are used to speed up search algorithms, such as depth-first search (DFS), on the original graph. The trade-off here is slower query times, but faster precomputation and minimal memory overhead. Since some of the algorithms presented in the following report very good query performance on some large graphs, but no experiments on time-expanded graphs have been evaluated so far, we hope for a good performance of such hybrid approaches. In the following, we will therefore evaluate some state-of-the-art Hybrid Labeling algorithms on our networks.

A number of such Hybrid Labeling algorithms have been developed, e.g., BFL [45], IP+ [46], PREACH [47], GRAIL [48], FERRARI [49] and TopChain [50]. Only TopChain [50] has been developed in the context of time-dependent route planning; the other algorithms have not been evaluated for route planning in public transport as far as we know. Some experiments for TopChain [50] were performed on GTFS instances, but no footpaths were modeled into the graph.

We used the original code of the publications and used the default parameters (as described in the respective papers). All algorithms operate on the time-expanded graph. In the case of TopChain, the departure and arrival events at each stop are organized into a chain decomposition, where a chain is defined as an ordered sequence of vertices such that each vertex can reach the next vertex in the chain [50]. In total, there are $|\mathcal{S}|$ many chains.

The query times shown in Table 5.7 represent the average time for 100 000 random event-to-event queries (not stop-to-stop). Similarly to PTL [16] and TopChain [50], any reachability oracle can be used to answer earliest arrival queries (cf. Section 3.3.2).

Answering one earliest arrival query requires a binary search over the arrival events at p_t, hence roughly

$$\left\lceil \log_2\left({}^{|\mathcal{E}|}/_{|\mathcal{S}|}\right) \right\rceil$$

reachability checks. In our networks ${}^{|\mathcal{E}|}/_{|\mathcal{S}|}$ ranges from about 70 to 173, so

$$\left\lceil \log_2\left({}^{|\mathcal{E}|}/_{|\mathcal{S}|}\right) \right\rceil \approx 7 \quad \text{(actual range } 6.1\text{-}7.4\text{)}$$

and thus about seven reachability queries per earliest arrival. This corresponds to the findings of [16].

However, these seven queries are always from the same start vertex, meaning it may be possible to store certain information between the runs in order to speed them up. For instance, in Hub Labeling, instead of computing the intersection of two labels $L_f(v_s), L_b(v_t)$ with a linear sweep over both labels, one can store $L_f(v_s)$ in a hashmap and simply check if any hub $h \in L_b(v_t)$ is stored in the hashmap [16]. Hybrid Labeling algorithms often combine search-based algorithms such as DFS with precomputed information to accelerate queries. During a search, these algorithms use preprocessed data to determine whether a path between the current vertex and the target exists; and prune if no path exist. However, since this information inherently depends on the target vertex, it is non-trivial to reuse it across multiple queries originating from the same source but toward different targets.

Note that the metrics in Table 5.7 report only the additional overhead introduced by the index structures; they exclude the memory consumption of the base graph itself (which is still needed for the query). PREACH needs to store the forward and backward graph, whereas the other algorithms only need the forward graph. All experiments were performed sequentially on the `Xeon` machine.

In terms of query performance, BFL and PREACH deliver the fastest results. BFL can answer reachability queries on Germany or Great Britain in less than $10\,\mathrm{ms}$ and $5\,\mathrm{ms}$, respectivly. While PREACH takes roughly double that time on those two networks, on smaller networks such as Switzerland, Paris and Berlin, it answers in less than $1\,\mathrm{ms}$. In contrast, IP+, GRAIL, and FERRARI show significantly slower query times, especially in larger networks where IP+ reaches up to $141\,\mathrm{ms}$ and GRAIL up to $75\,\mathrm{ms}$.

BFL dominates the preprocessing across the board, always at least a factor three faster than any other algorithm. PREACH requires moderate preprocessing time, while IP+, TopChain, and FERRARI often take considerably longer. As expected, good query performance costs memory, as can be seen with BFL and PREACH. GRAIL is by far the most memory-efficient algorithm, using only $222.27\,\mathrm{MB}$ on Germany and even less for smaller networks like Berlin $24.2\,\mathrm{MB}$. At the other end of the spectrum are BFL and PREACH, which are the most memory-intensive. BFL consumes over $2\,600\,\mathrm{MB}$ and PREACH consumes over $3\,100\,\mathrm{MB}$ in the German network. IP+, TopChain and FERRARI occupy the middle ground, offering reduced memory usage compared to BFL and PREACH, but requiring more memory than GRAIL.

Surprisingly, TopChain does not stand out in any metric, although the algorithm was developed for time-dependent routing. However, it was designed without the use of footpaths.

If we look at the fastest algorithms per network and multiply this by seven (as discussed above) to get a rough estimate of fixed departure time query performance, we get (in the order as listed in Table 5.7):

$$65.66\,\mathrm{ms}, 34.37\,\mathrm{ms}, 5.46\,\mathrm{ms}, 3.64\,\mathrm{ms} \text{ and } 1.54\,\mathrm{ms}.$$

The runtime corresponds approximately to the performance of CSA, but CSA does not need any precomputation time and additional memory overhead (cf. Table 5.5 and 5.6). Therefore,

it is clear that hybrid labeling approaches in the current form are not recommended for use on time-expanded graphs as black-box algorithm. Adapting hybrid labeling algorithms to the specific properties of public transit networks may offer promising opportunities for future research.

5.6.3 FLASH-TB and TB-CST

As expected, FLASH-TB and TB-CST deliver the fastest query times (cf. Table 5.5 and 5.6). FLASH-TB is the fastest algorithm on all networks, but also requires the most precomputation time. TB-CST, on the other hand, delivers similarly fast query times, but requires immense memory. However, both algorithms scale very poorly (both require quadratic overhead), so no experiments were conducted on the Europe network. T-REX, on the other hand, requires very little memory, takes seconds or minutes to precompute, but delivers query times orders of magnitude slower. Interestingly, T-REX queries on Germany are "only" a factor of 8.8 slower than TB-CST, but T-REX is three orders of magnitude better in terms of precomputation time and memory overhead compared to TB-CST.

As will be explained in more detail in Section 5.7.1, T-REX requires only one byte of additional memory per transfer, which is sufficient to handle up to 256 levels. Since FLASH-TB is configurable via the number of cells, we choose a partition such that the memory consumption is comparable between both algorithms. With an 8-partition for FLASH-TB, exactly one byte is stored per transfer. We now compare the query performance and preprocessing time of T-REX and FLASH-TB under equal memory consumption; cf. Table 5.8.

With comparable memory requirements, T-REX achieves query times on the German network similar to FLASH-TB, while preprocessing is approximately $2\,000$ times faster. Interestingly, FLASH-TB scans nearly twice as many trips but fewer transfers. This high number of scanned trips for FLASH-TB is likely caused by the *coning* effect [51] inherent to unidirectional Arc-Flags: outside the target cell, FLASH-TB primarily scans optimal journeys, but upon reaching the target cell, numerous "locally optimal transfers" are explored. Given the small number of cells, approximately $1/8$ of all stops lie within the target cell.

T-REX, which employs multiple hierarchical levels, is not affected *as much* by the coning effect, but still scans many trips because its LCL pruning is less effective than the Arc-Flag pruning of FLASH-TB. On the denser networks, T-REX scans more trips and transfers overall. Its query times are roughly twice as slow as those of FLASH-TB, although preprocessing remains substantially faster.

These results suggest that a simple border-event–based preprocessing, combined with a multilevel hierarchy, could serve as an effective precomputation strategy for FLASH-TB. The authors of FLASH-TB [5] note that it remains unclear whether a preprocessing approach that operates only within the "local" cell boundaries can still achieve competitive query performance. A hybrid approach that integrates the hierarchical structure of T-REX with

the cell-based techniques of FLASH-TB may therefore offer an attractive trade-off between query performance, preprocessing time, and memory consumption.

5.6.4 (Scalable) Transfer Patterns

Scalable Transfer Patters [18] was evaluated on a Germany instance, which with almost 15 million connections per day approximately corresponds to our Germany instance. However, our instance has twice as many stops, which will lead to a higher memory consumption, as the transfer patterns DAG has to be stored for each stop.

For Transfer Patterns, the authors report a precomputation time of $372\,\mathrm{h}$, with a memory consumption of $140\,\mathrm{GB}$. If the variant with hubs is used, which may deliver suboptimal results, the memory consumption is reduced to $10\,\mathrm{GB}$, the precomputation time to "only" $350\,\mathrm{h}$. The authors of [18, 52] mention that TP on the Germany instance needs about $0.3\,\mathrm{ms}$ for Pareto-optimal queries (using the Transfer Patterns variant without hubs).

For Scalable Transfer Patterns, the entire precomputation takes $16.5\,\mathrm{h}$ and a memory overhead of $1\,160\,\mathrm{MB}$. However, it should be noted that the authors do not mention anything about parallelization, which probably reduces the precomputation time considerably, since the calculation of the transfer patterns of individual stops can trivially be parallelized (see Section 5.7).

Regarding query times, the authors distinguish between "convex" local, "non-convex" local and global queries. The convex-local queries are very fast with $0.1\,\mathrm{ms}$, since the search space is also very small (on average there are 833 stations in a cluster). Non-convex-local queries require an average of $5\,\mathrm{ms}$. To draw a comparison, we also evaluated geo-rank queries for T-REX on the Germany instance, and for $r \in \{9, 10\}$, i.e., where the target stop is the 512-th or $1\,024$-th geographically distant stop from the source, a query takes 0.2 – $0.3\,\mathrm{ms}$. However, the size of a convex cluster does not have such a large impact on the query, since the transfer patterns depend only on the underlying structure of the network, not on the number of stops. Unfortunately, cells cannot be actively precomputed as convex; it is a by-product of the precomputation whether a cell is convex or non-convex. Therefore, it is unclear how many clusters are convex. The authors do not report this number either. For global queries, Scalable Transfer Patterns needs $32\,\mathrm{ms}$, whereas T-REX is a factor of 3.8 faster at $8.3\,\mathrm{ms}$. However, the authors cache the merged search graphs from global transfer patterns in order to avoid having to build up the search graph again and again. If this feature is not used, the query is slower and needs around $50\,\mathrm{ms}$.

Even on the significantly larger and undeniably more difficult European instance, T-REX is still faster on average than Scalable Transfer Patterns on the Germany instance ($18\,\mathrm{ms}$ vs $32\,\mathrm{ms}$). This difference is sufficiently large that variations in machine performance or minor implementation optimizations are unlikely to change the overall conclusion: T-REX is significantly faster than Scalable Transfer Patterns.

5.6.5 Comparison to CRP

CRP [8], a state-of-the-art algorithm for computing shortest paths on road networks, uses multilevel partitions to accelerate a bidirectional Dijkstra algorithm. This approach yields speedups of several orders of magnitude compared to the plain Dijkstra algorithm [14] on large-scale road networks such as those of Europe or North America.

Although the problem and the underlying graph are different, the question arises as to why the multilevel partition approach works so much better for road algorithms than for routing in public transport.

The exact reason is not entirely clear. However, a key factor is the strong hierarchical structure of road graphs. Empirical studies have shown that road networks exhibit small and balanced separators [21, 30, 53]. One possible explanation is their similarity to planar graphs, or the presence of natural cuts; that is, natural features such as mountain ranges or rivers [21, 40].

Since subgraphs of planar graphs are themselves planar, small and balanced separators can also be found recursively in these subgraphs. CRP benefits greatly from these properties: small separators imply fewer boundary vertices, which in turn results in quadratically fewer shortcuts within a cell and faster preprocessing. Furthermore, the balanced nature of the partitions ensures that fewer hierarchy levels are required to obtain very small cells. This, in turn, means that starting from a vertex, fewer shortcuts need to be evaluated to cover large portions of the graph, effectively reducing the unweighted diameter of the road network and enabling efficient traversal using only a limited number of shortcuts.

Public transport networks, on the other hand, do not exhibit such small and well-balanced separators. Although a certain hierarchy exists, for instance, between local public transport and long-distance services, it is far less pronounced than in road networks. Bast [36] notes that especially in metropolitan city centers, there is virtually no hierarchy, and thus little difference in importance between e.g., buses and trams. The first genuinely hierarchical level only emerges with long-distance transport between cities.

Another advantage of CRP is that it can run a bidirectional search from the source and target vertices simultaneously. In time-dependent networks, such as public transportation, the arrival time is unclear, and therefore backward searches are only possible to a limited extent.

Table 5.5: Average query times for uniformly random source–target stop pairs on large continent- and country-sized networks. See Subsection 5.5.1 for the experimental setup.

Network	Algorithm	Pareto	Query [μs]	Prepro. [hh:mm:ss]	k
Europe	TED	○	8 780 484	-	-
	TDD	○	1 270 411	-	-
	CSA	○	392 486	-	-
	ACSA	○	38 510	00:02:16	4 096
	RAPTOR	●	715 512	-	-
	TB	●	222 834	00:05:26	-
	T-REX	●	18 009	00:09:01	4 096
Germany	TED	○	1 466 681	-	-
	TDD	○	499 849	-	-
	CSA	○	83 101	-	-
	ACSA	○	18 333	00:00:10	512
	RAPTOR	●	212 255	-	-
	TB	●	71 030	00:00:45	-
	T-REX	●	8 331	00:00:58	512
	TB-CST^{s}	●	1 017	09:27:42	-
	FLASH-TB	●	127	32:47:49	8 192
Great Britain	TED	○	983 455	-	-
	TDD	○	170 510	-	-
	CSA	○	43 189	-	-
	ACSA	○	9 232	00:00:08	1 024
	RAPTOR	●	65 776	-	-
	TB	●	17 477	00:00:24	-
	T-REX	●	2 293	00:00:30	2 048
	TB-CST^{s}	●	191	02:13:13	-
	FLASH-TB	●	83	10:16:53	2 048
Switzerland	TED	○	95 992	-	-
	TDD	○	18 109	-	-
	CSA	○	3 931	-	-
	ACSA	○	1 899	00:00:02	256
	RAPTOR	●	10 719	-	-
	TB	●	4 605	00:00:04	-
	T-REX	●	924	00:00:06	1 024
	TB-CST^{p}	●	44	00:01:52	-
	FLASH-TB	●	15	00:06:43	8 192

Table 5.6: Average query times for uniformly random source–target stop pairs on the metropolitain networks. See Subsection 5.5.1 for the experimental setup.

Network	Algorithm	Pareto	Query [μs]	Prepro. [hh:mm:ss]	k
Paris	TED	○	249 700	-	-
	TDD	○	17 985	-	-
	CSA	○	3 280	-	-
	ACSA	○	6 301	00:00:16	64
	RAPTOR	●	8 562	-	-
	TB	●	3 746	00:00:07	-
	T-REX	●	2 018	00:00:18	1 024
	TB-CST$^{\text{p}}$	●	36	00:04:00	-
	FLASH-TB	●	25	00:19:42	16 384
Berlin	TED	○	74 524	-	-
	TDD	○	16 310	-	-
	CSA	○	2 415	-	-
	ACSA	○	2 484	00:00:03	128
	RAPTOR	●	7 259	-	-
	TB	●	3 055	00:00:04	-
	T-REX	●	1 046	00:00:05	512
	TB-CST$^{\text{p}}$	●	49	00:01:08	-
	FLASH-TB	●	33	00:05:20	16 384

Table 5.7: Performance of the different Hybrid Labeling algorithms on the time-expanded graphs. Shown are the average query times for random vertex-to-vertex reachability queries, the total sequential preprocessing times and the overall memory overhead for the index data structures.

Metric	Network	BFL	IP+	PREACH	TopChain	GRAIL	FERRARI
Query [ms]	Germany	**9.38**	116.35	23.02	35.21	64.31	46.28
	Great Britain	**4.91**	141.59	10.20	24.25	75.14	62.88
	Switzerland	1.93	13.74	**0.78**	1.61	11.87	5.40
	Paris	2.08	5.42	**0.52**	1.08	5.75	2.16
	Berlin	1.18	7.83	**0.22**	1.28	3.74	3.33
Preproc. [s]	Germany	**31.74**	112.58	87.67	105.43	83.75	120.03
	Great Britain	**15.40**	60.42	53.37	58.35	51.25	70.70
	Switzerland	**3.27**	11.17	10.72	12.13	12.23	12.46
	Paris	**3.89**	19.85	19.69	15.81	13.40	16.66
	Berlin	**2.10**	7.00	7.17	7.96	7.76	7.53
Memory [MB]	Germany	2 666.34	1 434.24	3 111.00	2 886.96	**222.27**	1 498.71
	Great Britain	1 756.29	731.83	2 049.00	1 902.22	**146.37**	1 002.27
	Switzerland	431.11	179.72	503.47	466.89	**35.96**	236.44
	Paris	404.67	235.62	472.18	438.25	**33.73**	231.55
	Berlin	290.08	120.92	338.74	314.14	**24.20**	165.63

Table 5.8: The table shows the query time, number of scanned trips and relaxed transfers averaged over 10 000 random fixed departure time queries. Additionally, the total precomputation time, including transfer generation, is reported. For the FLASH-TB algorithm, the number of cells is set to 8, while for T-REX, the best-performing configuration of the number of levels and imbalance is used, similar to Tables 5.5 and 5.6.

Network	Algorithm	Query [μs]	Scan. Trips	Rel. Transfers	Prepro. [hh:mm:ss]
Germany	T-REX	8 331	17 341	531 454	00:00:58
	FLASH-TB	8 737	32 084	395 224	32:44:04
Switzerland	T-REX	924	4 160	61 968	00:00:06
	FLASH-TB	454	3 025	22 397	00:14:17
Paris	T-REX	2 018	7 099	197 045	00:00:18
	FLASH-TB	940	3 253	76 122	00:19:17

5.7 Memory–Query–Preprocessing Trade-Offs

We now also compare memory consumption and preprocessing times versus the achieved query performance of the algorithms for Pareto-optimal journeys, cf. Figures 5.9 and 5.10. The dataset is Germany, and the data for FLASH-TB and TB-CST are taken from the paper [5]. Since we do not have access to any code from Scalable Transfer Patterns and Transfer Pattern, we have taken the reported numbers from their paper [18, 52]. It should also be mentioned here that the Germany dataset is not exactly the same, but the number of connections is roughly comparable. However, our instance has about twice as many stops, which means that more (and possibly larger) transfer patterns have to be saved. Nevertheless, we expect the query times to be roughly the same.

5.7.1 Memory Consumption

See Figure 5.9. T-REX only needs to store the cell ID for each stop, as well as the rank for each transfer, i.e., in our implementation a 16-bit unsigned integer for the cell ID and an 8-bit unsigned integer for the rank. Technically, since we only evaluate T-REX up to 16 levels, the rank could be stored using just four bits per transfer. If desired, it would be possible to combine two 4 bit ranks into a single byte, reducing memory usage at the cost of additional bit manipulation during query execution. However, because the smallest addressable unit in memory is eight bits, we simply use one byte per transfer.

The memory overhead compared to TB is linear in the number of stops and transfers, and thus on Europe needs only about $515\,\mathrm{MB}$, Germany $113\,\mathrm{MB}$, Great Britain $60\,\mathrm{MB}$, Switzerland $15\,\mathrm{MB}$, Paris $44\,\mathrm{MB}$ and Berlin $11\,\mathrm{MB}$.

As expected, increased memory consumption generally results in improved query performance. T-REX achieves a favorable memory–performance balance and clearly outperforms Scalable Transfer Patterns. With only a minimal memory overhead compared to TB, T-REX already achieves an order of magnitude of query speedup, and closes the gap between TB and FLASH-TB to a certain extend.

5.7.2 Precomputation Time

We have also plotted the precomputation time versus query performance in Figure 5.10. The precomputation time is the parallel time with 128 threads for T-REX, TB, FLASH-TB, and TB-CST. For Scalable TP and TP (without hubs), we assumed ideal parallel speedup for the precomputations. The reported sequential preprocessing times of $16.5\,\mathrm{h}$ [18] and $372\,\mathrm{h}$ [52] were divided by 128, resulting in estimated parallel preprocessing times of approximately $0.129\,\mathrm{h}$ (≈ 7.7 minutes) and $2.91\,\mathrm{h}$ ($\approx 02{:}55$ hh:mm), respectively. This provides a lower bound for the actual parallel precomputation time, as a perfect speedup is very unlikely. Furthermore, both precomputations depend linearly on the number of

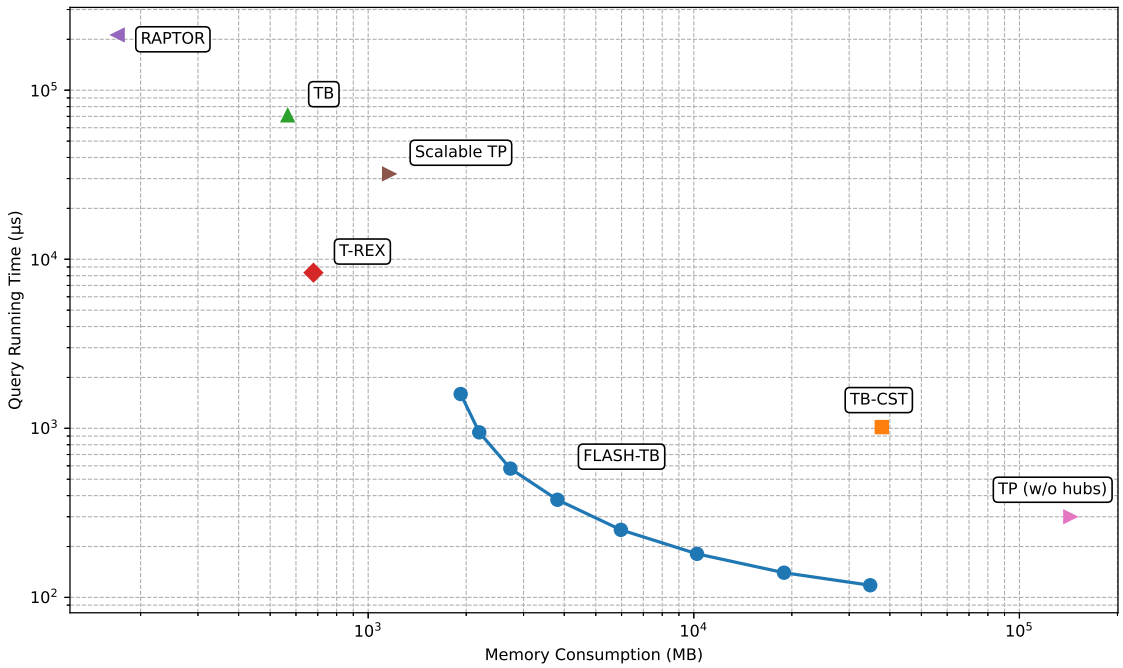


Figure 5.9: The memory usage and query times for Pareto-optimal fixed-departure-time algorithms on the Germany network are shown (both axes are logarithmic). Lower-left is best (minimal memory, fastest queries). The memory consumption for all TB-based algorithms includes the memory consumption of the underlying TB. FLASH-TB and TB-CST^s data are from [5], with FLASH-TB plotted at varying number of cells (fewer cells mean less memory but slower queries). Scalable Transfer Patterns and TP (without hubs) metrics are taken from [18, 52] on a similarly sized Germany instance (see Section 5.7) and should be compared with caution.

stops, and our Germany instance has almost twice as many stops as their reported instance. RAPTOR does not require any additional precomputation time as it operates directly on the timetable, hence it is not shown in the Figure.

FLASH-TB requires the most precomputation time, regardless of the number of cells. TB-CST and TP also require several hours, although the times reported for TP are optimistic estimates. Our own implementation of TP confirms the observation from [7]: TP without hubs takes more precomputation time than TB-CST without split trees. Scalable TP, by contrast, is slower than T-REX in precomputation (even with optimistic estimates) and also yields slower query times.

Overall, T-REX performs very well among state-of-the-art algorithms. Both FLASH-TB and T-REX seem to dominate other state-of-the-art algorithms in the trade-off between memory usage and query performance, lying between the two extremes: RAPTOR (minimal memory, slower queries) and PTL (very high memory, extremely fast queries).

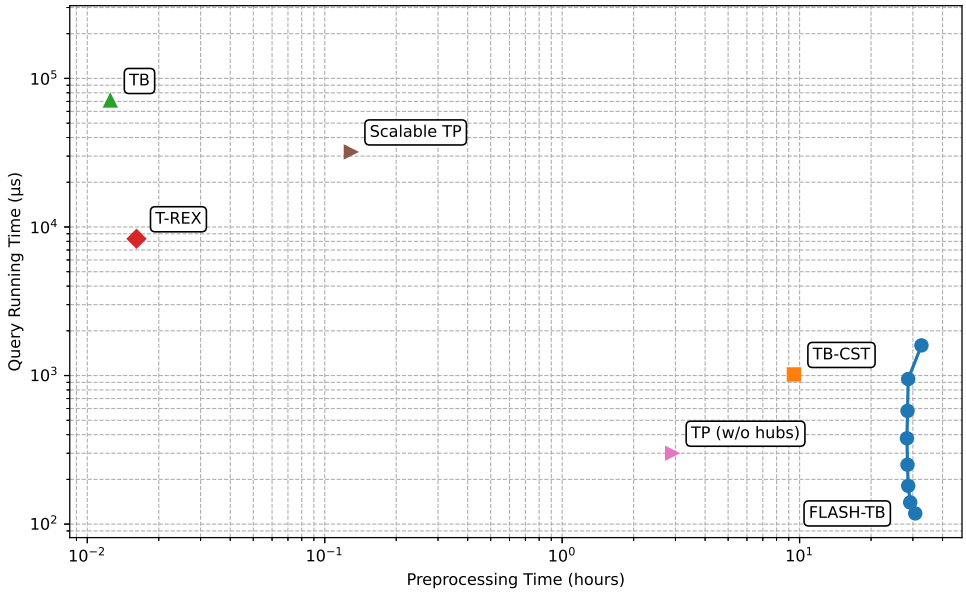


Figure 5.10: The preprocessing times and query running times for Pareto-optimal fixed-departure-time algorithms on the Germany network are shown (both axes are logarithmic). Lower-left is best (minimal preprocessing time, fastest queries). FLASH-TB and TB-CST^s data are from [5], with FLASH-TB plotted at varying numbers of cells. Scalable Transfer Patterns and TP (without hubs) metrics are estimated for a *perfect* 128-thread speedup based on the sequential preprocessing times reported in [18, 52] on a similarly sized Germany instance (see Section 5.7). These values should be interpreted as approximate and likely overoptimistic, since perfect or near-perfect parallel speedup is unlikely.

6 Conclusion

We presented T-REX, a new route planning algorithm based on multilevel partitioning and the state-of-the-art Trip-Based Public Transit Routing [1] (TB) algorithm. The new algorithm can answer queries in less than $19\,\mathrm{ms}$ even on continent-sized networks, requiring linear memory overhead and only a few minutes of precomputation time. Unlike previous partition-based journey planning algorithms, T-REX achieves a remarkable speedup over TB not only on continent- and country-sized networks such as Europe and Germany, but also on metropolitan regions such as Paris or Berlin. The query times are not as fast as those of e.g., FLASH-TB [5], but are fast enough for realistic reactive applications. T-REX demonstrates that multilevel approaches can be effectively combined with TB. Storing information about transfers and leveraging it during query execution to restrict the search space proves to be highly effective. The overall results suggest that it is worthwhile to take a closer look at multilevel approaches and to analyze why T-REX, despite its simple and fast preprocessing, achieves such strong performance.

6.1 Future Work

The next step is to show the correctness of the two conjectures 4.5.5 and 4.5.6, in order to prove T-REX. An interesting aspect that is relevant for use in practice is the handling of timetables that span an arbitrary period of time, using e.g., the TB presented in this paper [37]. A customization with *all* IBEs would be wasteful, so one would either have to set the customization to a fixed time frame (e.g., only consider the current operating day), or adapt the customization for such time spans with frequency-based techniques [52]. In addition, an adaptation would be necessary to be able to deal efficiently with unrestricted footpaths, for which an approach in combination with ULTRA (UnLimited TRAnsfers) [22, 54] could be interesting. Moreover, it would be interesting to develop special graph partitioners for public transportation networks to not only improve performance and precomputation time, but to get good results on all types of public transit networks. Achieving this would likely require a deeper understanding of the structural properties of public transit networks. In addition, similar to ACSA, arbitrary multilevel partitions could be evaluated, e.g., three or five cells per level. The bit-efficient calculation of the LCL is not trivial in this case, but could justify the additional overhead if the search space can be significantly reduced. A potential direction for future work is the introduction of shortcuts across cells instead of marking individual transfers and assigning a rank. The memory consumption will probably

increase and the query algorithm will have to be adapted. However, the query times will probably be better, as one no longer has to process a long transfer path within a large cell, but instead skip an entire cell in one step. As discussed in Section 5.6.3, future work could investigate whether the preprocessing of T-REX can be adapted to generate the flags required by FLASH-TB more efficiently, without sacrificing query performance.

A More Experiments

A.1 T-REX Metrics

Table A.1: Listed are average query metrics of T-REX and ACSA per level on the Europe network. For T-REX, we report the number of scanned trips, relaxed transfers, query time and preprocessing. 10 000 random queries were evaluated using the fixed departure time queries method. For level zero, we report the metrics for TB and CSA. Preprocessing for T-REX does not include the time to generate the transfers. Query times are shown as [μs], preprocessing as [mm:ss]. For every level, we choose the imbalance that lead to the best query performance. Referenced in Section 5.5.2.

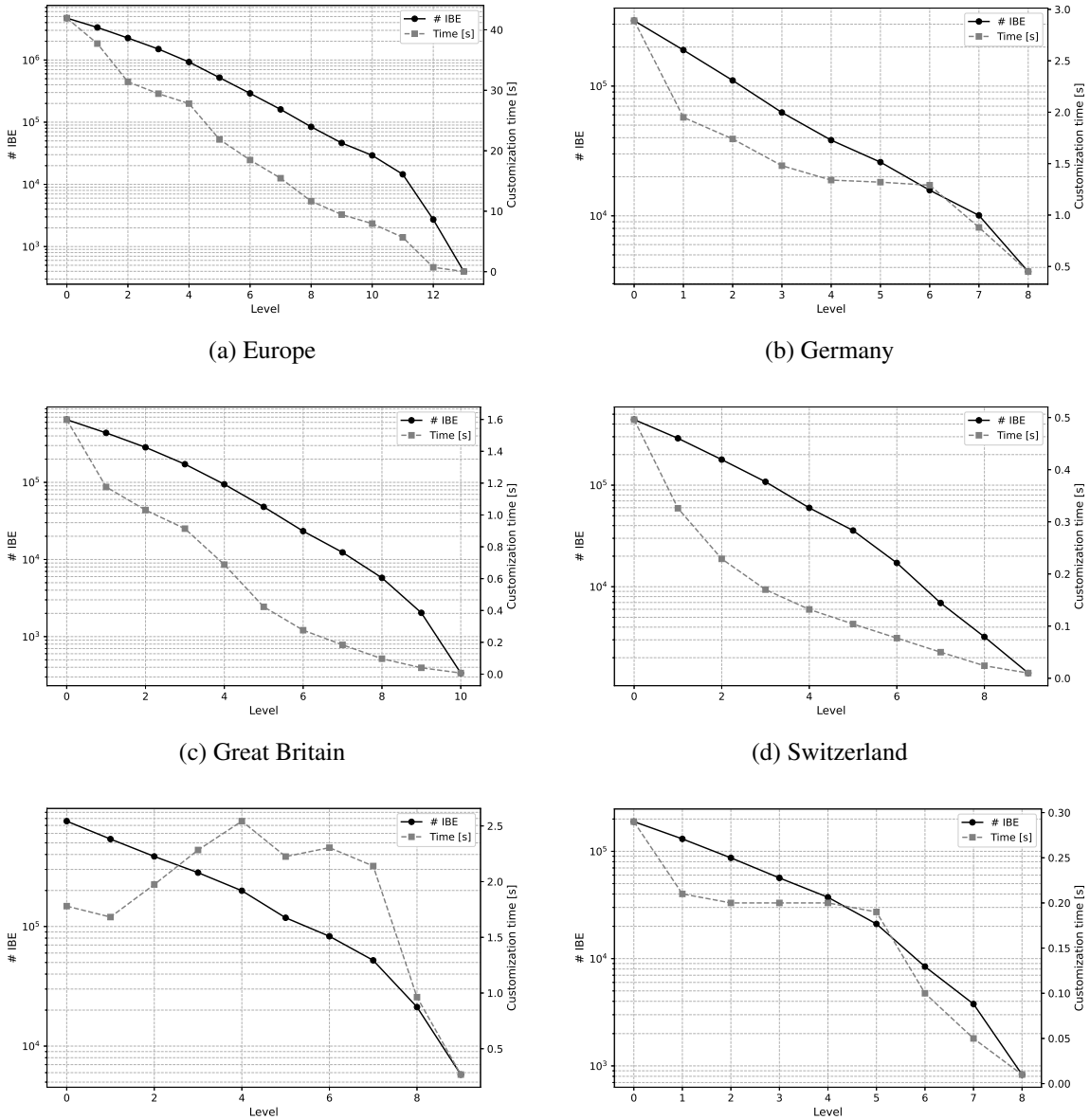
Levels	Scan. Trips	Rel. Transfers	T-REX		ACSA	
			Query	Prepro.	Query	Prepro.
0	750 177	14 648 126	207 452	–	392 486	–
7	39 815	1 458 167	23 637	**01:16**	71 808	**00:56**
8	31 995	1 362 692	23 201	01:31	52 477	00:57
9	31 480	1 320 013	24 592	01:32	41 673	01:14
10	24 847	1 242 960	19 196	02:24	**38 289**	01:30
11	23 780	1 227 393	18 868	03:08	39 083	02:01
12	23 313	1 228 190	**18 009**	03:36	38 510	02:16
13	22 948	1 191 870	19 934	04:49	39 520	02:52
14	**22 876**	**1 173 682**	19 594	04:20	38 567	03:59
15	23 776	1 183 777	19 666	03:19	58 055	05:27
16	23 642	1 228 332	20 894	05:40	59 965	08:16

Table A.2: Listed are average query metrics of T-REX and ACSA per level on the Great Britain (above) and Switzerland (below) network. For T-REX, we report the number of scanned trips, relaxed transfers, query time and preprocessing. 10 000 random queries were evaluated using the fixed departure time queries method. For level zero, we report the metrics for TB and CSA. Preprocessing for T-REX does not include the time to generate the transfers. Query times are shown as $[\mu s]$, preprocessing as $[mm{:}ss]$. For every level, we choose the imbalance that lead to the best query performance. Note that on the Switzerland instance, some customization times were below half a second (marked by $< 00{:}01$, instead of 00:00). Referenced in Section 5.5.2.

Levels	T-REX				ACSA	
	Scan. Trips	Rel. Transfers	Query	Prepro.	Query	Prepro.
0	79 054	1 388 371	17 477	–	43 137	–
5	11 297	251 555	3 792	**00:01**	27 222	**00:05**
6	6 923	178 965	2 829	00:02	17 680	**00:05**
7	5 919	162 756	2 581	00:02	13 058	00:06
8	4 862	147 230	2 399	00:04	10 303	00:06
9	4 641	145 545	2 363	00:05	9 402	00:07
10	4 443	141 710	2 304	00:06	**9 232**	00:08
11	**4 466**	**140 021**	**2 293**	00:06	10 117	00:08
12	4 489	143 091	2 323	00:09	9 879	00:11
0	32 577	309 660	4 629	–	3 931	–
4	7 632	91 431	1 387	00:01	3 812	**00:01**
5	5 667	73 077	1 123	**< 00:01**	3 022	**00:01**
6	4 900	67 228	1 016	00:01	2 415	00:02
7	4 598	64 572	974	**< 00:01**	2 098	00:02
8	4 376	64 570	930	00:01	**1 899**	00:02
9	4 381	63 386	949	00:01	2 028	00:03
10	**4 160**	**61 968**	**924**	00:02	**1 932**	00:03
11	4 316	64 348	935	00:02	1 974	00:03

Table A.3: Listed are average query metrics of T-REX and ACSA per level on the Paris (above) and Berlin (below) network. For T-REX, we report the number of scanned trips, relaxed transfers, query time and preprocessing. 10 000 random queries were evaluated using the fixed departure time queries method. For level zero, we report the metrics for TB and CSA. Preprocessing for T-REX does not include the time to generate the transfers. Query times are shown as [μs], preprocessing as [mm:ss]. For every level, we choose the imbalance that lead to the best query performance. Referenced in Section 5.5.2.

Levels	T-REX				ACSA	
	Scan. Trips	Rel. Transfers	Query	Prepro.	Query	Prepro.
0	20 574	422 123	3 746	–	**3 280**	–
5	7 734	205 978	2 075	**00:04**	6 406	**00:13**
6	7 374	199 582	2 019	00:06	6 301	00:16
7	7 473	203 975	2 072	00:08	6 784	00:21
8	7 245	198 198	2 043	00:08	6 814	00:22
9	**7 076**	**196 533**	2 034	00:10	7 761	00:26
10	7 099	197 045	**2 018**	00:11	7 808	00:28
0	21 302	256 373	3 055	–	**2 415**	–
5	5 331	83 521	1 189	**00:01**	2 926	**00:03**
6	4 383	74 357	1 085	**00:01**	2 524	**00:03**
7	4 201	72 762	1 060	**00:01**	2 484	**00:03**
8	4 320	73 686	1 066	**00:01**	2 600	**00:03**
9	**4 164**	**72 403**	**1 046**	**00:01**	2 843	00:04
10	4 359	74 034	1 057	**00:01**	2 865	00:04



(a) Europe

(b) Germany

(c) Great Britain

(d) Switzerland

(e) Paris

(f) Berlin

Figure A.1: Number of incoming border events (IBEs) and the runtime of the customization algorithm per level. For each dataset, the configuration (number of levels and imbalance) that yielded the best query performance was chosen; cf. Tables 5.5 and 5.6. Note: the left y-axis (the number of IBEs) is shown on a logarithmic scale. Referenced in Section 5.4.

A.2 Hub Labeling Experiments

We evaluated two parallel hub labeling algorithms designed for reachability queries on DAGs. The parallelization scheme introduced in [55], developed by the author as part of a seminar, enables efficient parallel computation of reachability labels on DAGs. This scheme was applied to two established hub labeling algorithms: PLL [56] (Pruned Landmark Labeling) and PPL [57] (Pruned Path Labeling). The corresponding implementations are publicly available[1][2].

The results on the evaluated networks are presented below. Query performance was measured as described in Subsection 5.6.2, using the average running time over $100\,000$ random event-to-event queries. Both algorithms were preprocessed on the `Epyc` machine with 128 threads, utilizing 512-bit SIMD instructions. Query times were evaluated on the `Xeon` machine. Query times shown in parentheses indicate measurements obtained on the `Epyc` machine due to higher memory requirements.

Referenced in Section 5.6.2.

Table A.4: The table presents the results of the parallel PPL algorithm, including the average event-to-event query times, the average sizes of the forward and backward labels, as well as the total memory consumption and preprocessing time.

Network	Query [μs]	Avg. $\lvert L_f \rvert$	Avg. $\lvert L_b \rvert$	Memory [MB]	Prepro. [hh:mm:ss]
Europe	(1.43)	103.16	98.59	170 429	09:22:52
Germany	1.12	120.52	129.59	57 147	01:24:55
Great Britain	0.88	69.19	70.20	22 744	00:22:26
Switzerland	0.61	56.70	49.79	4 405	00:02:25
Paris	0.87	120.41	116.53	8 531	00:05:31
Berlin	0.70	63.85	59.90	3 381	00:01:11

[1] `https://github.com/PatrickSteil/DAG-HL`
[2] `https://github.com/PatrickSteil/PPL`

Table A.5: The table presents the results of the parallel PLL algorithm, including the average event-to-event query times, the average sizes of the forward and backward labels, as well as the total memory consumption and preprocessing time.

Network	Query [μs]	Avg. $\lvert L_f \rvert$	Avg. $\lvert L_b \rvert$	Memory [MB]	Prepro. [hh:mm:ss]
Germany	(1.97)	284.71	230.01	167 952	04:53:50
Great Britain	1.31	164.84	141.11	66 941	01:56:37
Switzerland	0.81	136.94	116.93	13 653	00:05:49
Paris	2.14	507.49	515.82	49 951	00:13:52
Berlin	1.16	206.77	192.94	14 447	00:03:02

Bibliography

[1] Sascha Witt. Trip-based Public Transit Routing. *CoRR*, abs/1504.07149, 2015.

[2] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. 2016.

[3] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-based Public Transit Routing. *Transp. Sci.*, 49(3):591–604, 2015.

[4] Ernestine Großmann, Jonas Sauer, Christian Schulz, and Patrick Steil. Arc-Flags Meet Trip-based Public Transit Routing. *CoRR*, abs/2302.07168, 2023.

[5] Ernestine Großmann, Jonas Sauer, Christian Schulz, Patrick Steil, and Sascha Witt. FLASH-TB: Integrating Arc-flags and Trip-based Public Transit Routing. *CoRR*, abs/2312.13146, 2023.

[6] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast Routing in Very Large Public Transportation Networks Using Transfer Patterns. In Mark de Berg and Ulrich Meyer, editors, *Algorithms - ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part I*, volume 6346 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2010.

[7] Sascha Witt. Trip-based Public Transit Routing Using Condensed Search Trees. *CoRR*, abs/1607.01299, 2016.

[8] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning in Road Networks. *Transp. Sci.*, 51(2):566–591, 2017.

[9] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable Contraction Hierarchies. *CoRR*, abs/1402.0402, 2014.

[10] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection Scan Algorithm. *CoRR*, abs/1703.05997, 2017.

[11] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Tobias Zündorf. Faster Transit Routing by Hyper Partitioning. In Gianlorenzo D'Angelo and Twan Dollevoet, editors, *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, ATMOS 2017, September 7-8, 2017, Vienna, Austria*, volume 59 of *OASIcs*, pages 8:1–8:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[12] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM J. Exp. Algorithmics*, 5:12, 2000.

[13] Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-criteria Shortest Paths in Time-dependent Train Networks. In Catherine C. McGeoch, editor, *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, volume 5038 of *Lecture Notes in Computer Science*, pages 347–361. Springer, 2008.

[14] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[15] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental Study on Speed-up Techniques for Timetable Information Systems. In Christian Liebchen, Ravindra K. Ahuja, and Juan A. Mesa, editors, *ATMOS 2007 - 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems, November 15-16, 2007, Sevilla, Spain*, volume 7 of *OASIcs*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[16] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. Public Transit Labeling. *CoRR*, abs/1505.01446, 2015.

[17] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[18] Hannah Bast, Matthias Hertel, and Sabine Storandt. Scalable Transfer Patterns. In Michael T. Goodrich and Michael Mitzenmacher, editors, *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016*, pages 15–29. SIAM, 2016.

[19] Hannah Bast, Jonas Sternisko, and Sabine Storandt. Delay-robustness of Transfer Patterns in Public Transportation Route Planning. In Daniele Frigioni and Sebastian Stiller, editors, *13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, ATMOS 2013, September 5, 2013, Sophia Antipolis, France*, volume 33 of *OASIcs*, pages 42–54. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

[20] Prateek Agarwal and Tarun Rambha. Scalable Algorithms for Bicriterion Trip-based Transit Routing. *CoRR*, abs/2111.06654, 2021.

[21] Aaron Schild and Christian Sommer. On balanced separators in road networks. In Evripidis Bampis, editor, *Experimental Algorithms - 14th International Symposium, SEA 2015, Paris, France, June 29 - July 1, 2015, Proceedings*, volume 9125 of *Lecture Notes in Computer Science*, pages 286–297. Springer, 2015.

[22] Jonas Sebastian Sauer. *Closing the Performance Gap Between Multimodal and Public Transit Journey Planning*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2024.

[23] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. Efficient route planning on public transportation networks: A labelling approach. In Timos K.

Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 967–982. ACM, 2015.

[24] Robert Geisberger. Contraction of Timetable Networks with Realistic Transfers. In Paola Festa, editor, *Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*, volume 6049 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 2010.

[25] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In Catherine C. McGeoch, editor, *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2008.

[26] Arno Eigenwillig. An Update on Fast Transit Routing with Transfer Patterns — research.google. `https://research.google/blog/an-update-on-fast-transit-routing-with-transfer-patterns/`. [Accessed 19-08-2024].

[27] Arthur Finkelstein and Jean-Charles Régin. Using goal directed techniques for journey planning with multi-criteria range queries in public transit. In Greg H. Parlier, Federico Liberatore, and Marc Demange, editors, *Proceedings of the 10th International Conference on Operations Research and Enterprise Systems, ICORES 2021, Online Streaming, February 4-6, 2021*, pages 347–357. SCITEPRESS, 2021.

[28] Vassilissa Lehoux and Christelle Loiodice. Faster Preprocessing for the Trip-based Public Transit Routing Algorithm. In Dennis Huisman and Christos D. Zaroliagis, editors, *20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, ATMOS 2020, September 7-8, 2020, Pisa, Italy (Virtual Conference)*, volume 85 of *OASIcs*, pages 3:1–3:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[29] Thomas Papadakis. *Skip lists and probabilistic analysis of algorithms*. PhD thesis, CAN, 1993. UMI Order No. GAXNN-81138.

[30] Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. Faster and better nested dissection orders for customizable contraction hierarchies. *Algorithms*, 12(9):196, 2019.

[31] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world'networks. *Nature*, 393(6684):440–442, 1998.

[32] Amine Abou-Rjeili and George Karypis. Multilevel algorithms for partitioning power-law graphs. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006.

[33] Lars Gottesbüren and Michael Hamann. Deterministic parallel hypergraph partitioning. In *European Conference on Parallel Processing (Euro-Par)*, volume 13440, pages 301–316. Springer, 2022.

[34] Lars Gottesbüren. *Parallel and Flow-Based High-Quality Hypergraph Partitioning*. PhD thesis, Karlsruhe Institute of Technology, 2023.

[35] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. Scalable high-quality hypergraph partitioning. *ACM Transactions on Algorithms*, 20(1):9:1–9:54, 2024.

[36] Hannah Bast. Car or Public Transport - Two Worlds. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 355–367. Springer, 2009.

[37] Sascha Witt. Extending the Time Horizon: Efficient Public Transit Routing on Arbitrary-length Timetables. *CoRR*, abs/2109.14143, 2021.

[38] Patrick Steil. Optimal FIFO grouping in public transit networks. *CoRR*, abs/2308.06629, 2023.

[39] Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013.

[40] Daniel Delling, Andrew V. Goldberg, Ilya P. Razenshteyn, and Renato Fonseca F. Werneck. Graph Partitioning with Natural Cuts. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*, pages 1135–1146. IEEE, 2011.

[41] Peter Sanders and Christian Schulz. Distributed Evolutionary Graph Partitioning. In David A. Bader and Petra Mutzel, editors, *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments, ALENEX 2012, The Westin Miyako, Kyoto, Japan, January 16, 2012*, pages 16–29. SIAM / Omnipress, 2012.

[42] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way hypergraph partitioning via n-level recursive bisection. In Michael T. Goodrich and Michael Mitzenmacher, editors, *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016*, pages 53–67. SIAM, 2016.

[43] Mo Chen, Rezaul Alam Chowdhury, Vijaya Ramachandran, D L Roche, and Lin Tong. Priority Queues and Dijkstra's Algorithm. 2007.

[44] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Robust Exact Distance Queries on Massive Networks. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 321–333. Springer, 2014.

[45] Jiao Su, Qing Zhu, Hao Wei, and Jeffrey Xu Yu. Reachability querying: Can it be even faster? *IEEE Trans. Knowl. Data Eng.*, 29(3):683–697, 2017.

[46] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. Reachability querying: an independent permutation labeling approach. *VLDB J.*, 27(1):1–26, 2018.

[47] Florian Merz and Peter Sanders. Preach: A fast lightweight reachability index using pruning and contraction hierarchies. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 701–712. Springer, 2014.

[48] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. GRAIL: a scalable index for reachability queries in very large graphs. *VLDB J.*, 21(4):509–534, 2012.

[49] Stephan Seufert, Avishek Anand, Srikanta J. Bedathur, and Gerhard Weikum. FERRARI: flexible and efficient reachability range assignment for graph indexing. In Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1009–1020. IEEE Computer Society, 2013.

[50] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. Reachability and time-based path queries in temporal graphs. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 145–156. IEEE Computer Society, 2016.

[51] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. In J. Ian Munro and Dorothea Wagner, editors, *Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments, ALENEX 2008, San Francisco, California, USA, January 19, 2008*, pages 13–26. SIAM, 2008.

[52] Hannah Bast and Sabine Storandt. Frequency-based search for public transit. In Yan Huang, Markus Schneider, Michael Gertz, John Krumm, and Jagan Sankaranarayanan, editors, *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Dallas/Fort Worth, TX, USA, November 4-7, 2014*, pages 13–22. ACM, 2014.

[53] Michael Hamann and Ben Strasser. Graph bisection with pareto optimization. *ACM J. Exp. Algorithmics*, 23, 2018.

[54] Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. ULTRA: Unlimited Transfers for Efficient Multimodal Journey Planning. *Transportation Science*, 57:1536–1559, 2023.

[55] Patrick Steil. Parallel PLL on DAGs, 2025.

[56] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 349–360. ACM, 2013.

[57] Takuya Akiba. Pruned labeling algorithms: fast, exact, dynamic, simple and general indexing scheme for shortest-path queries. In *Proceedings of the 23rd International*

Conference on World Wide Web, WWW '14 Companion, page 1339–1340, New York, NY, USA, 2014. Association for Computing Machinery.